

A MULTITASKING AND DATA-DRIVEN ARCHITECTURE FOR MULTI-AGENTS SIMULATIONS

THÈSE N° 3545 (2006)

PRÉSENTÉE LE 4 JUILLET 2006

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

Laboratoire de réalité virtuelle

SECTION D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Sébastien SCHERTENLEIB

ingénieur informaticien diplômé EPF
de nationalité suisse et originaire de Vechigen (BE)

acceptée sur proposition du jury:

Prof. C. Petitpierre, président du jury
Prof. D. Thalmann, directeur de thèse
Prof. A. Camurri, rapporteur
Prof. A. de Antonio Jiménez, rapporteur
Prof. M. Odersky, rapporteur



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Lausanne, EPFL

2006

ABSTRACT

The expansion of 3D real-time simulations (3DRTS) into millions of homes together with the technical progress of computers hardware force to approach software developments for 3DRTS from different perspectives. From an historical standpoint, 3DRTS started principally as homebrew developments. The underlined consequences are the lack of standardization for producing such applications. Nowadays, computers hardware can reproduce close to photo-realism 3D images within interactive environments. This was made possible with the continuous improvements in computers hardware. During many years, the hardware evolution was following vertical speed-up improvements, by increasing CPU clocks speed and memory bandwidth. Today, we are reaching the limits of this approach from a power consumption, heat, and intrinsic materials characteristics perspectives.

As an outcome, the next-generation of computer hardware and home consoles are presenting multitasking architectures. This obliges to re-think software development for 3DRTS, moving from the serial and single-threaded approach to a concurrent design. We explore conceptual designs handling the current scale and complexity offered by 3DRTS developments by adopting stronger engineering practices. This is needed to control the underlined complexity and rising developments costs. The direct consequence of being able to generate highly detailed virtual worlds is to involve more deeply artists and designers in the development process. We propose mechanisms that free developers from common low-levels problematic, such as memory management or data synchronization issues. Our architecture relies on extending the Component Based Development (CBD) model for multitasking architectures. This obliges to define specific patterns either directly inspired by other fields in computer science or dedicated for 3DRTS. This includes promoting multi-layer design where the low-level routines are tightly connected to computer hardware by describing the importance of conceiving hardware-oblivious systems. This is important, as memory bandwidth is becoming the principal bottleneck in current applications. Another fundamental aspect consists to move from the single iterative global loop commonly found in single-threaded systems, by incorporating mechanisms for balancing the workflow more accurately.

If those optimizations and evolutions are required for assuring efficient real-time performance, they do not allow non-programmers to interact with the system with ease. Our method consists to promote high-level languages and concurrent model relying on Microthreads. This gives the ability to develop and execute scripts in a multitasking environment without the common C/C++ issues. This is primordial to let designers experiment with ideas in a safer and efficient environment. This will leads to adopt the data-driven paradigm to control agents in our simulations, by clearly separating the logic and data layers. This offer better flexibility and reduce the existence of simulation specific code. In addition, we illustrate that the best technology and designs have a limited meaning, if they do not come with a complete production pipeline for managing and controlling simulation assets. This also affects fine tuning parameters where different hardware may perform better in some areas or worse in other. Finally, different use-cases demonstrate the strong and weakness aspects of our approach.

Keywords: Software Engineering, 3D Graphics, Data-Driven, Concurrency.

RESUME

L'intégration de simulations 3D temps réel dans des millions de foyers associés avec les progrès techniques dans le matériel informatique force d'approcher la conception de logiciels sous d'autres angles. D'un point de vue historique, le développement d'applications 3D temps réel a principalement débuté au travers de développements individuels. La conséquence directe est le manque de standardisation pour la production de telles applications. De nos jours, les ordinateurs sont capables de reproduire des images 3D photo réaliste dans des environnements interactifs. Cela a été possible grâce au développement continu du matériel informatique. Durant des années, l'évolution du matériel suivait une amélioration verticale en augmentant la fréquence des processeurs ainsi que des transfert des données. Aujourd'hui, nous atteignons les limites de cette approche pour des raisons de consommation d'énergie, de dégagement de chaleur, ainsi que des caractéristiques intrinsèques des matériaux.

En conséquence, les futures générations de PCs et de consoles de jeux présentent une architecture multi-tâches. Ceci oblige à re-penser le processus de développement pour les applications 3D temps réel, en modifiant l'approche classique et sérielle, par une approche concurrente. Nous explorons les concepts permettant de manager la taille et la complexité des applications 3D temps réel en adoptant de plus strictes méthodes d'ingénierie. Ceci est absolument nécessaire pour contrôler, à la fois la complexité et les coûts de développements. La conséquence directe étant d'être en mesure de générer des mondes virtuels détaillés afin d'intégrer les artistes et designers dans le processus de développement. Nous proposons des mécanismes qui libèrent les développeurs des habituels problèmes de bas niveaux, tels que la gestion de la mémoire ou la synchronisation des données. Notre architecture repose dans l'extension du modèle «Component Based Development» (CBD) pour des architectures multi-tâches. Ceci requière de définir des patrons spécifiques, soit directement inspiré d'autres domaines ou directement dédié aux applications 3D temps réel. Ceci inclus aussi de promouvoir un design multicouche où les fonctions de bas niveaux sont directement connectées aux matériels en décrivant l'importance de concevoir des systèmes capables d'exploiter le matériel à disposition. Ceci est primordial, car les transferts de données sont le principal goulet d'étranglement dans les applications actuelles. Un autre aspect fondamental consiste à ne plus se reposer sur une approche itérative en y incorporant des mécanismes pour répartir les tâches.

Si toutes ces optimisations et évolutions sont requises pour assurer des performances optimales, elles ne permettent pas aux personnes non qualifiées d'interagir facilement avec le système. Notre méthode consiste à promouvoir des langages de hauts niveaux ainsi que des modèles de programmations concurrentes basées sur le concept de Microthreads. Cela offre l'opportunité de développer et d'exécuter des scripts dans un environnement multi-tâches en évitant les problèmes du C/C++. C'est primordial pour permettre aux designers d'expérimenter leurs idées dans un environnement sécurisé et efficace. Ceci entraîne l'adoption de méthodes permettant le contrôle des personnages virtuels aux travers des données, en séparant la logique et les données. Cela accroît la flexibilité et réduit l'existence de code spécifique à une simulation. De plus, nous illustrons que les meilleures technologies et designs, n'ont qu'un intérêt limité, sans l'accompagnement d'outils de productions efficaces. Ceci inclus des outils de paramétrisations permettant d'adapter les simulations aux différents matériaux existants. Finalement, certains exemples démontrent les points forts et faibles de notre approche.

Mots-clés : Technologie de la programmation, Graphismes 3D, Programmation dirigée par les données, Programmation concurrente.

TABLE OF CONTENTS

Chapter 1 Introduction.....	1
1.1 Domain.....	1
1.2 Motivation.....	2
1.3 Assumptions and Challenges	2
1.4 Approaches	3
1.5 Structure of the Thesis	3
Chapter 2 Concepts and Terms.....	5
2.1 Architecture.....	5
2.2 Interactive Simulation Systems.....	5
2.2.1 Visualization.....	6
2.2.2 Character Animations	7
2.2.3 Level of Detail (LOD)	7
2.2.4 Load-Balancing	7
2.2.5 Scripting Systems	8
2.2.6 Artificial Intelligence (AI).....	8
2.3 Middleware	8
2.3.1 Middleware Components.....	9
2.3.2 3D APIs	9
2.3.3 Digital Content Creation (DCC) Tools.....	10
2.3.4 Environments Editors	10
2.3.5 3D Modeling Software	11
2.4 Methodologies for Systems Architectures	11
2.4.1 Technical Design	12
2.4.2 Collaborative Work	12
2.4.3 Code Maintainer	13
2.4.4 Conventions.....	13
2.4.5 Assets Management.....	13
2.4.6 Source Control.....	13
2.4.7 Automate Build System.....	14
2.4.8 Unit-Testing Framework	14
2.5 Conclusion	14
Chapter 3 Related Work	15
3.1 System Architecture.....	15
3.1.1 Software Engineering for Interactive Simulations.....	16
3.1.2 Architecture Design Methodology	17
3.1.3 System Architectures Categories.....	21
3.1.4 3DRTS Systems Evolution.....	21
3.2 VR Systems.....	23
3.2.1 Distributed VR Systems	23
3.2.1.1 DIVE.....	23
3.2.1.2 dVS	24
3.2.1.3 MORGAN.....	24
3.2.2 Toolkit.....	24
3.2.2.1 SVE.....	24
3.2.2.2 World Tool Kit.....	25
3.2.3 Object-Oriented Frameworks	25
3.2.3.1 ALICE.....	25
3.2.3.2 MAVERICK	25

3.2.3.3 JADE.....	25
3.2.3.4 DIVERSE.....	26
3.2.3.5 LIGTHNING	26
3.2.3.6 VIRPI.....	27
3.2.3.7 VR Juggler	28
3.2.3.8 Virtools	29
3.2.4 Component-Based Frameworks.....	30
3.2.4.1 I4D	30
3.3 Game Systems.....	31
3.3.1 VR Systems vs. Games Systems	31
3.3.2 3D Graphic Rendering Centric Engines	31
3.3.2.1 Ogre3D.....	32
3.3.2.2 Torque Game Engine	32
3.3.2.3 Crystal Space	32
3.3.3 Component-Based Game Framework.....	33
3.3.3.1 RenderWare	33
3.3.3.2 Unreal Technology.....	35
3.4 Conclusion	36
Chapter 4 Scope and Structure of Work	37
4.1 Design Principles	37
4.1.1 The Development Process	38
4.1.2 Framework Development Complexity	38
4.1.3 Software Architecture Layers	39
4.1.4 Non Stable Hardware Problematic	39
4.1.5 The Non-Design Oriented Approach.....	40
4.1.6 Misconceptions and Solutions to Prevent Them.....	40
4.1.7 Enforcing Standards and Conventions.....	41
4.1.8 Documentation	41
4.2 Architecture.....	42
4.2.1 Objectives.....	42
4.2.2 Build for People.....	43
4.2.3 Hardware Scalability	43
4.2.4 Software Scalability.....	44
4.2.4.1 Scaling the Environment.....	44
4.2.4.2 Scaling 3D Models.....	44
4.2.4.3 Scaling Character Animation	45
4.2.4.4 Scaling Special Effects.....	45
4.2.4.5 Scaling Methodology	46
4.2.5 Adding Variety	46
4.2.5.1 Variety in Textures	46
4.2.5.2 Variety in Behaviors	47
4.2.6 Reusability	47
4.2.6.1 Software Reusability Validation	48
4.3 Conclusion	48
Chapter 5 Component-Based Software.....	49
5.1 Component Model.....	49
5.1.1 Component Based Architecture	50
5.1.2 Multi-Layers Architecture	50
5.1.3 Abstraction	51
5.1.4 Loose Coupling	52
5.1.5 Dependency Graph	52
5.1.6 Keeping Localized Dependencies.....	53

5.1.7	Indirection	53
5.1.8	Orthogonality.....	53
5.1.9	Object-Oriented Programming Methodology	54
5.1.10	Productivity	54
5.1.11	Investigating Existing 3DRTS	54
5.2	VHD++ Architecture.....	55
5.2.1	Physical Source Lines of Code (SLOC)	56
5.2.2	Choosing a Topology	56
5.3	Concrete Implementation	57
5.3.1	Intrinsic Functionalities	57
5.3.2	Low-Level Layer	57
5.3.3	Intermediate Layer.....	58
5.3.4	High-Level Layer	58
5.4	vhdService.....	58
5.5	Object Manager.....	59
5.5.1	Properties Manager.....	61
5.5.2	Interactive Simulation Object Management	61
5.6	Data-Driven Architecture.....	62
5.6.1	Logic Layer	63
5.6.2	Data Layer	63
5.6.3	Data Manipulation	63
5.6.4	Interactive Simulation Objects (ISO) Structures	63
5.6.4.1	Object-Centric vs. Component-Centric.....	63
5.6.4.2	Basic Object.....	65
5.6.4.3	vhdProperty.....	65
5.6.5	Decoupling Information	66
5.6.6	Communication Interfaces.....	66
5.6.7	Data and State Controller	67
5.6.8	Late Binding Mechanisms	67
5.6.8.1	Dynamic Component Management.....	68
5.6.8.2	XML Files.....	68
5.6.9	Scripting	69
5.6.10	Flow of Data.....	69
5.7	Data Streaming.....	69
5.8	Serialization	70
5.9	Scene Representation	71
5.9.1	Related Work.....	72
5.9.2	Meta Graph Manager.....	72
5.9.3	MultiView Scene Representation	74
5.10	Architectural Patterns.....	75
5.10.1	Concurrent Design Patterns	75
5.10.2	Concurrent Micro-Kernel	76
5.11	Multithread Models.....	76
5.11.1	CPU and GPU Analogy	77
5.11.2	Concurrency Design	77
5.11.3	Designing with Regards to the Dependency Graph.....	77
5.12	Performance and Optimizations	78
5.13	Conclusion	78
Chapter 6	System and Technology	79
6.1	CPU.....	79
6.1.1	Multi-Processors Architectures	80
6.1.2	Current Limitations	80
6.1.3	Technology Trends.....	81
6.1.4	Benefits and Costs	81

6.1.5 Performance Gain	82
6.1.6 Tools	83
6.2 Technology Awareness	83
6.2.1 Concurrent Models	83
6.2.2 Multi Core Systems	83
6.2.3 Multi Cores Systems and OS	84
6.2.4 Threading API	84
6.2.5 Synchronicity	85
6.2.5.1 Synchronization at Component Level	85
6.2.5.2 Synchronization at Object Level	86
6.2.5.3 Choosing a Synchronization Topology	86
6.2.6 OpenMP	86
6.3 Cache Memory	87
6.3.1 Cache Coherence	87
6.3.2 Spatial and Time Locality	88
6.3.3 Future Hardware and Cache Memory	89
6.4 Concurrent Model	89
6.4.1 Non-Determinism Software	90
6.4.2 Concurrency Specific Issues	90
6.4.3 Modular Parallelism	90
6.4.4 The Parallelism Categories	91
6.4.5 Micro Concurrent Model	92
6.4.5.1 Applying Synchronous Computations	92
6.4.5.2 Managing H-ANIM Human Animation in Parallel	92
6.4.6 Task Scheduling	93
6.5 Cache-Oblivious Algorithms	93
6.5.1 Computer Memory Architecture	94
6.5.1.1 Data Structure and Cache Coherence	94
6.5.1.2 Van Emde Boas Layout	94
6.6 Cache-Oblivious Hierarchical Tree	95
6.6.1 ABT-Tree	95
6.6.1.1 A Ram Implementation	95
6.6.1.2 ABT Tree Creation	96
6.6.1.3 Efficiency	96
6.6.1.4 Complexity	97
6.6.1.5 Exploiting Redundancy	97
6.6.1.6 Performance	99
6.6.1.7 Results	100
6.7 Low-Level Libraries Optimizations	100
6.8 Multi Layer Concurrent Programming Techniques	102
6.9 Concurrent System and Memory Management	103
6.10 Tasks Dispatching	104
6.11 Data Synchronization	105
6.12 System Control Flow	106
6.13 Managing Processing Throughput with Concurrent Systems	107
6.14 Conclusion	108
Chapter 7 Scripting	109
7.1 History	109
7.2 Interpreted vs. Compiled Languages	109
7.3 Related Work	109
7.4 The Python Interpreter and Component-Based Software	110
7.4.1 Benefits of using a Scripting Language	110
7.4.2 Disadvantages of Scripting Languages	111
7.4.3 A Proper Usage	111

7.4.4 The Choice of Python	112
7.4.5 Code Bindings	112
7.4.6 Python Module Registration	114
7.4.7 Performance	114
7.4.8 Memory Management	114
7.5 Concurrent Multitasking	115
7.5.1 Coroutines and Microthreads	115
7.5.2 Microthread Manager	115
7.5.2.1 Implementation	116
7.5.3 Embedding Python	117
7.5.3.1 Blocking Mode	117
7.5.3.2 Cooperative Multitasking Mode	117
7.5.3.3 Concurrent Multitasking Mode	118
7.5.3.4 Custom Preprocessor and Specific Keywords	118
7.5.3.5 Threading Management	119
7.6 Authoring Tool	120
7.6.1 Features	120
7.6.2 vhdPythonService XML Configuration	121
7.6.3 Late Bindings	121
7.7 Special Use Case	121
7.7.1 Safe Sand Box	122
7.8 Optimization	122
7.8.1 Garbage Collector	122
7.8.2 Stackless Python	122
7.8.3 JIT Compiler	123
7.8.4 Byte-Code	123
7.9 Experiments	123
7.10 Conclusion	124
Chapter 8 Data-Driven AI Engine	125
8.1 History of 3D Real-Time AI	125
8.1.1 Restrictive AI Techniques	126
8.2 AI Engine	126
8.2.1 Core Principle of AI Systems	127
8.2.2 Related Work	127
8.2.3 AI Engine Architecture	128
8.2.3.1 System Overview	128
8.2.3.2 Behavioral Engine	128
8.2.3.3 Data-Driven Classes and Properties	129
8.2.3.4 Hierarchical FuSM (Fuzzy State Machine)	130
8.2.3.5 C++ and Lua Metatable Bindings	131
8.2.3.6 AI Architecture Using Prioritized Tasks	132
8.3 Behavior	134
8.4 Variety	135
8.5 Authoring	135
8.5.1 Offline Pipeline	135
8.5.2 XML Configuration	136
8.5.3 Runtime Information	137
8.6 Performance	137
8.6.1 Tasks Ordering	138
8.6.2 Level of Detail (LOD)	138
8.7 Conclusion	138
Chapter 9 Authoring	139

9.1	Unlocking System Potential	139
9.2	Object-Editing Toolkit	139
9.3	Assets Management	141
9.4	Content Creation Pipeline	141
9.5	Exporters	142
9.6	GUIs for Interactivity	143
9.7	Input Device (Gamepad)	143
9.8	VHD++ Technology Map	144
9.9	Profiling	145
9.9.1	Commercial Profiling Tools	145
9.9.2	Custom in-Simulation Profiler	146
9.9.3	Profiler User Interfaces	147
9.9.4	Profiler API	147
9.9.5	Profiling and Performance issues	147
9.9.6	Improving Performance using Scalability Tests-Units	148
9.10	Conclusion	148
Chapter 10	Case Studies	149
10.1	JUST	150
10.1.1	JUST VR System	151
10.1.2	Scenario Authoring	152
10.2	Virtual Orchestra	153
10.2.1	Managing 3D Sound Processing	154
10.2.2	Synchronization of Animations with Sound Propagation	155
10.2.3	Input Devices: a Virtual Orchestra Controlled by a Real Human	155
10.3	LIFEPLUS (AR Simulation)	157
10.3.1	AR Life System Design	157
10.3.2	AR Life Tracking	158
10.3.3	MR Framework Operation for Character Simulation	159
10.3.4	Threading Model	159
10.3.5	MR Registration and Staging	160
10.3.6	Interactive Objects Manipulation	161
10.3.7	Hardware Setup	162
10.4	Ogre3D Integration	163
10.5	Autonomous Virtual Human in Persistent Worlds	164
10.5.1	Running Simulation into VHD++	165
10.6	ERATO (Crowds)	167
10.6.1	Resources Management	168
10.6.2	Managing the Simulation Workflow	169
10.6.3	Scenario Prototyping	170
10.6.4	Scenario Management	170
10.6.5	Simulation Controls	172
10.7	vCrowds	173
10.7.1	Navigation Graph	174
10.7.2	Animation Variety	175
10.7.3	Multithreading for Crowd Simulations	175
10.8	Other Use Cases	177
10.9	Conclusion	178
Chapter 11	Conclusion	179
11.1	Summary	179
11.2	Contributions	179
11.3	Discussion and Lessons Learned	180
11.4	Future Topic	180

11.5 Epilogue	181
Bibliography	183
Appendix A Existing Platforms and Their Associated Standards	199
A.1 OpenGL	199
A.2 DirectX	199
A.3 OpenGL ES	199
A.4 UML	200
A.5 XML	200
A.6 X3D	200
A.7 CONTRIGA	201
A.8 XDS	201
A.9 Collada	202
A.10 H-ANIM	202
A.11 XNA	203
A.12 OpenInventor	204
A.13 OpenSG	204
A.14 OpenSceneGraph	205
A.15 OpenAL	205
A.16 FMod	205
A.17 ODE	206
A.18 CEGUI	206
A.19 Doxygen	206
Appendix B Hardware Architecture for 3DRTS	207
B.1 Evolution of multi-core CPUs	207
B.2 Mainstream PCs and PC Workstations	208
B.3 Hyper-Threading	208
B.4 XCPU	209
B.5 CELL	210
B.6 GPU Architecture	211
B.7 Physic Processing Unit (PPU) Architecture	212
Appendix C Cross-Platforms Considerations	213
C.1 Component-Based System and Cross-Platforms Projects	213
C.2 Content Creation Pipeline and Cross-Platforms Projects	214
C.3 Metadata Files	214
C.4 Loading Time Consideration	215
C.5 Binary Files Formats	215
C.6 Memory Fragmentation	217
Appendix D VHD++ Run-Time GUIs	219
Appendix E XML System Configuration File	223
Appendix F XML Data Configuration Files	225
Appendix G Python Scripts	229
Appendix H Lua Scripts	233
Appendix I Build System	239

Appendix J Posters	247
--------------------------	-----

LIST OF FIGURES

Figure 2.1: The evolution in 3D graphics showcased by the Unreal Engine.	6
Figure 2.2: A virtual human and its skeleton.	7
Figure 2.3: The middleware layer provides a software abstraction between the application and the platform specific operating systems and hardware.	8
Figure 2.4: 3D graphics developers have to choose between the OpenGL and DirectX APIs.	9
Figure 2.5: In the left, the Quark editor. In the middle the UnrealEd editor and in the right the Hammer editor.	10
Figure 2.6: 3D Modeling tools allow creating 3D geometry within a dedicated environment (on the left: Softimage, on the right: Maya).	11
Figure 2.7: Pyramid of the development process methodologies [Keith2006].	12
Figure 3.1: The quality improvement of interactive simulations over the time (here a selection of some major video games).	16
Figure 3.2: Typical architecture used in most 2D games in 1994 [Blow2004].	21
Figure 3.3: Early 3D real-time applications were still mixing 2D with 3D elements [Blow2004].	22
Figure 3.4: Modern 3DRTS systems contain many components that need to communicate in an efficient manner [Blow2004].	22
Figure 3.5: A typical MMO architecture design [Blow2004].	23
Figure 3.6: DIVERSE libraries encapsulation.	26
Figure 3.7: The LIGHTNING system architecture.	27
Figure 3.8: Objects are directly coupled with system functions.	27
Figure 3.9: The VIRPI, Aura, and CAVESStudy interconnections.	28
Figure 3.10: VR Juggler multi-layer system architecture.	29
Figure 3.11: The Virtools components workflow.	29
Figure 3.12: The left picture represent the I4D design and development flowchart. The right picture describes the component layers featured by the I4D system architecture.	30
Figure 3.13: RenderWare multi-layer architecture.	33
Figure 3.14: RenderWare studio overview.	34
Figure 3.15: Software dependencies use in the video games Gears of War [Sweeney2006].	35
Figure 3.16: Unreal Technology threading model.	36

Figure 4.1: Relative software development complexity as presented in The Mythical Man-Month book.	39
Figure 4.2: Coding style.	41
Figure 4.3: Mipmap allow optimizing the texturing performance by reducing the memory footprint required for rendering distant objects.	44
Figure 4.4: Increasing 3D mesh geometry using subdivision surface methodology.	45
Figure 4.5: The image on the left show the game running a single core processor, while the image on the right run on a dual core CPU, where one core is use for eye candy special effects [Davies2005].	45
Figure 4.6: Variety in textures.	46
Figure 4.7: The base variety texture displayed along the alpha map in the middle and the triangle laid out in the right. The square beneath the alpha map show the alpha colors.	47
Figure 4.8: Base variety texture (most left) and the alpha map (next to it) with some combination of colors.	47
Figure 5.1: Components and the simulation specific layer.	50
Figure 5.2: Multi-layers separation.	51
Figure 5.4: Cyclic-Dependencies between A, B and C.	53
Figure 5.5: VHD++ Architecture.	55
Figure 5.7: Mixed topologies using on demand components implementation.	56
Figure 5.8: vhdService implementation.	59
Figure 5.9: Object manager hierarchy.	60
Figure 5.10: Properties manager.	61
Figure 5.11: Static classes' hierarchy.	64
Figure 5.13: Example of a “ <i>Diamond of Death</i> ” shape.	64
Figure 5.14: Dynamic classes' hierarchy.	64
Figure 5.15: The concrete data and states controller is separate from the underlined system controller, which only keep a memory representation of the considered simulation object.	67
Figure 5.16: Data loading from storage devices.	70
Figure 5.18: The Meta Graph Manager.	73
Figure 5.23: Concept of a vhdAnimGraph where each node is shared (vhdNodeProperty).	73
Figure 5.25: Concept of a vhdRenderGraph, optimized for fast rendering.	74
Figure 5.26: Concept of a multi-view scene representation.	75

Figure 5.27: On the top, a single threaded application using the parallel processing power of modern GPU. On the bottom, a multithreaded application, that dedicated one thread for the CPU/GPU communication.....	77
Figure 6.1: Serial code impact in parallel speedup values.....	82
Figure 6.2: OpenMP task dispatching for two threads.....	87
Figure 6.3: Memory cache layers attaches to the CPU (Registers, L1 and L2 cache) and the main memory.	87
Figure 6.4: Performance Analyzer Tool [SCEE2002].....	88
Figure 6.5: Workflow execution using the framework concurrency model.	89
Figure 6.6: The different parallelism categories.....	91
Figure 6.7: Characters animation updates are spread over several threads.....	93
Figure 6.8: Memory multi-level hierarchy.	94
Figure 6.9: Van Emde Boas's tree representation. In this example, each subtree is composed of 7 nodes.	94
Figure 6.10: a) the original scene b) KD-Tree organization c) ABT-Tree organization.	95
Figure 6.11: Axis aligned bounding box readjustment.....	96
Figure 6.12: Searching time order.	97
Figure 6.13: ABT tree's node data representation (couple).	98
Figure 6.14: Cache line organization.....	98
Figure 6.15: Subtree organization with linking to following subtrees.....	99
Figure 6.17: Memory usage across implementations. The memory consumption for the cache-oblivious depends on the tree's balancing.	100
Figure 6.18: Testbed applications for analyzing and validating our ABT tree implementation.....	100
Figure 6.19: Main aspect-graph composed of vhdProperty serving as a concurrent access synchronization layer to access data underneath.....	102
Figure 6.20: Every thread see the repartition of memory zones into three categories: invalid memory, dedicated memory and shared memory.....	103
Figure 6.21: The component's update execution is spread over different threads (here 4).	104
Figure 6.22: Data synchronization pipeline between components.....	105
Figure 6.23: Inter-threads communication with data synchronization.....	106
Figure 6.24: The main three control flow layers.....	107
Figure 6.25: Processing throughput management for concurrent systems.....	108

Figure 7.1: Microthreads execution model.....	116
Figure 7.2: Case 1, blocking mode.	117
Figure 7.3: Case 2, cooperative multitasking.....	118
Figure 7.4: Case 3, concurrent multitasking.	118
Figure 7.5: Python Console.	120
Figure 7.6: Performance impact.	124
Figure 9.1: On the left part the Unreal Kismet Editor. On the right, the Unreal scripting console.	139
Figure 9.2: On the left, the Neverwinter Nights toolkit, and the right the Elder Scroll III: Morrowind toolkit.....	140
Figure 9.3: OSGEdit allow to edit graphical object properties and to analyze the internal scene graph.	140
Figure 9.4: The hierarchy for content management pipeline.	141
Figure 9.5: Content creation pipeline.	142
Figure 9.6: On the left, set of widgets for interacting with virtual humans, on the right crowd brush to dynamically affect emotion to virtual characters.	143
Figure 9.7: The FEdit Editor for creating force feedbacks effects files (.ffe).....	143
Figure 9.8: The left analog stick move the character, the right analog stick allows turning the character.	144
Figure 9.9: The VHD++ technology map.....	145
Figure 9.10: Diagnostic and profiling widgets.	147
Figure 10.1: Spectrum of applications.....	149
Figure 10.2: JUST emergencies simulations.	150
Figure 10.3: JUST VR system concept.....	151
Figure 10.4: JUST VR navigation paradigm: “navigation ring” metaphor allowing for “walking around” the virtual environment; “sliding margin” metaphor allowing for “looking around” the virtual environment.	152
Figure 10.5: JUST VR system scenario execution: the main GUI used by the simulation supervisor to direct interactive scenario execution.....	153
Figure 10.6: On the left, a virtual flutist following the user’s tempo. On the right, an end-user interacts with the system using a PDA acting as an input device in front of a big projector screen.....	154
Figure 10.7: Example of a sound path depending of external events that control the synchronization between the sound rendering and the human animation.	155
Figure 10.8: The handheld interface and interprets the data from the magnetic tracker.....	156

Figure 10.9: LIFEPLUS: augmented reality scenario.	157
Figure 10.10: The mobile AR-life simulator system (left). On the right, the real-time virtual Pompeian characters in the real site of the Pompeian thermopolium. Note the use of geometry ‘occluders’ that allow part of the real scene to occlude part of the virtual human (right).....	158
Figure 10.11: AR component pipeline.....	158
Figure 10.12: Threading model.	160
Figure 10.13: Projection and model view matrix transformation from the real camera to the VR scene.	161
Figure 10.14: Virtual humans grasping virtual objects.....	162
Figure 10.15: AR life simulator system.....	163
Figure 10.16: Simulations using Ogre3D for the 3D renderer.....	164
Figure 10.17: On the top left: the 3D window. On the bottom left: the navigation graph. On the right: controller for observing the evolution of motivation and set of triggers to change priorities dynamically.....	165
Figure 10.18: Different viewpoints from the Odeon and the crowds.	167
Figure 10.19: Representing and positioning social classes in the Odeon.	168
Figure 10.20: Animation blending pipeline.....	169
Figure 10.21: Simulation workflow that show the separation between the rendering and artificial intelligence execution threads.	170
Figure 10.22: Interaction with the crowd is possible through using predefined scenarios and camera viewpoints.....	172
Figure 10.23: Global application overview.	173
Figure 10.24: Crowd simulation within an urban environment.....	174
Figure 10.25: On the left, the circus with its 5 entries and on the right, its nodes (delimited with blue) connected with the rest of the graph only at these 5 entry points.....	174
Figure 10.26: Navigation FSM: the path is composed of sub-goals based on the navigation graph.	175
Figure 10.28: Threading workflow.....	176
Figure 10.29: CPU/GPU synchronization points.....	176
Figure 10.30: Collection of use cases and examples.	177
Figure A.1: The two OpenGL ES profiles: the Common Profile, and the Safety Critical Profile.	200
Figure A.2: The three layers of hierarchy found in the CONTRIGA standard.....	201
Figure A.3: Collada integration pipeline.	202

Figure A.4: Subset of the H-ANIM standard commonly used by H-ANIM compliant characters.	203
Figure A.5: The OpenSG is built around four modules separating low levels and high level of interactions.	204
Figure A.6: OpenSceneGraph Plug-in encapsulation.	205
Figure A.7: CEGUI widgets are independent of the underlined 3D API. It can work with existing 3D engines and pure OpenGL or DirectX applications.	206
Figure B.1: Evolution of Multi-core CPUs [Plumb2006].	207
Figure B.2: The dual-core architecture design used by AMD for its 64-bits X86 processors.	208
Figure B.3: Projection of tie-ratio of PCs with multi-core CPUs.	208
Figure B.4: The left part represent the Hyper-Threading architecture design used by Intel in its Pentium IV line of processors. On the right, the same processor without Hyper-Threading.	209
Figure B.5: The XCpu communications layer between the three cores unit and the crossbar unit.	210
Figure B.6: The CELL architecture featuring seven active SPU units and one PPU unit.	211
Figure B.8: GPU pipelined architecture [Kiel2006].	212
Figure B.10: PPU Block Diagram.	212
Figure C.1: System components and its underlined platform specific implementations.	214
Figure C.2: Memory usage of the same structure based on data alignment constraints.	216
Figure C.3: Memory fragmentation.	217
Figure C.4: The different memory allocators.	218
Figure D.1: vhdWidgets providing diagnostic information about the running system.	219
Figure D.2: vhdWidgets for run-time interaction.	220
Figure D.3: Diagnostic and profiling widgets.	221
Figure J.1: Cultural heritage.	247
Figure J.2: Illustration of the action selection for autonomous virtual humans from Etienne de Sevin.	248
Figure J.3: Augmented reality simulations.	249
Figure J.4: ERATO project overview.	250
Figure J.5: VHD++ platform overview.	251

LIST OF TABLES

Table 3.1: Major modules of 3D real-time simulations.....	21
Table 4.1: Software development priorities.....	43
Table 5.1: Categorizing components.....	55
Table 5.2: SLOC statistics.....	56
Table 5.3: SLOC comparison chart.....	56
Table 5.4: Intrinsic Mechanisms.....	57
Table 5.5: Extract of vhdServices.....	58
Table 6.1: Parallelism layers among technical constraints.....	91
Table 9.1: In-Simulation profiler functionalities comparison.....	146
Table 10.1: All defined motivations with associated actions and their locations.....	166
Table 10.3: Scenario workflow.....	171
Table C.1: Loading time.....	215

GLOSSARY

3DRTS: Cross-section of Game, VR, AR applications focusing interactive, real-time 3D simulations involving synthetic characters usually in the context of virtual storytelling.

Cache Memory: Collection of data duplicating original values stored elsewhere, where the original data (in term of access time) is more expensive. Once a data is in cache, further data access is accelerating since the cache average access time is lower.

Cache-Oblivious: Algorithms where no variable depend on hardware parameters such as cache-size and cache-line length, while achieving optimal performance in hardware independent way. Cache-Oblivious algorithm generally outperform RAM algorithm.

Concurrency: Property of systems that execute stream of instructions in parallel and which may permit the sharing of common resources. Some side effects include race conditions that can result in unpredictable system behavior or problems such as deadlock and starvation.

Crowd: A large number of persons gathered together.

Data-Driven: In computer science, data-driven design is the result of separating both the logic and data layer allowing uploading new content without changing the logic layer.

Design Pattern: A general repeatable solution to a commonly occurring problem in software design.

Embedded Platforms: Fix and stable computers hardware featuring identical performance and features (e.g. home consoles, handheld devices...). The contrary of evolutive platforms such as PCs.

Lua Metatable: A metatable is an ordinary Lua table, but with extra specifications to its behavior (and how it uses user supplied data) under certain operations. In our case, they represent dedicated states such as specific scenario-based rules. They also provide reference access to the C++ character entity data structure.

Mission-Critical: Software that is highly dependent on the underlined hardware for the good application behavior. Typical examples are 3DRTS, which are mostly constrained by the hardware RAW performances. This term is popular for describing applications that required optimizing resources for real-time computations.

Multitasking: This refers to the abilities to handle multiple tasks simultaneously. Modern CPUs and GPUs are highly multitasking processing units.

Multi-core: A multi-core computer combines two or more independent processors into a single package often a single integrated circuit. In general, multi-core computer allow exhibiting some form of thread-level parallelism without including multiple microprocessors in separate physical package.

Microthreads: "User" thread, which involve transfers of control. A microthread represents a unit of execution that has its own private data and control stack, while sharing global variables with other microthreads. Due to a cooperative concurrency model, microthreads must explicitly request to be suspended before another microthread may run.

RAW: Often, this term refer to present the theoretical and/or expected performance throughput of computer hardware. In the general circumstance, the measured values are considerably lower than RAW values.

Systems Architectures: Represent a software system. This is a representation, because it conveys the information content and the relationship between elements as well as the rules governing those relationships. They are also a process defining a sequence of steps and constraints. Finally, it is a discipline that requires a body of knowledge or general principles of software architecture to inform developers to the most effective way to design a system within a set of constraints.

Variety: The quality or state of having different forms or types.

vhdYIELD: Specific keyword based on the Python yield statement. This keyword allows suspending the function's execution, preserving the local variables.

LIST OF ABBREVIATION

3Dc	Compression algorithm for normal map textures
3DRTS	3D Real-Time Simulations
ABT	Adaptive Binary Tree
AI	Artificial Intelligence
ANSI	American National Standard Institute
API	Application Programming Interface
AR	Augmented Reality
ASCII	American Standard Code for Information Interchange
CAS	Commercial Available Software
CBD	Component Based Development
COTS	Commercial-Off-The-Shelf
CPU	Computer Processing Unit
DCC	Digital Content Creation
DXTn	Compression algorithm for mip-maping texture
EU	European Union
FPS	Frame Per Second (updates per second in context of RT audio-visual simulations)
FPS (gender)	First-Person Shooter
FSM	Finite State Machine
FuFSM	Fuzzy Finite State Machine
GPU	Graphical Processing Unit
HFSM	Hierarchical Finite State Machine
HMD	Head-Mounted Display
ID	IDentifier
I/O	Input/Output
ISO	Interactive Simulation Object
ISOUID	Interactive Simulation Object Unique IDentifier
LOD	Level Of Detail
MMO	Massively Multiplayer Online
NDI	Non Developmental Item
NET	Net Present Value
OO	Object Oriented
OOP	Object Oriented Programming
OSG	OpenSceneGraph
OpenGL ES	OpenGL for Embedded Systems
OS	Operating System
PC	Personal Computer
R&D	Research & Development
RAW	Read After Write
RAM	Random Access Memory
RPG	Role Playing Game
RTTI	Run-Time Type Information
SCEA	Sony Computer Electronic of America
SDK	Software Development Kit
SISD	Single Instruction Single Data
SIMD	Single Instruction Multiple Data
SKU	Software Kit Unit
SLOC	Source Lines of Code
SMT	Simultaneous MultiThreading
SSE	Streaming SIMD Extension
TDD	Test-Driven Development
UML	Unified Modeling Language
VHD	Virtual Humans Director
VR	Virtual Reality
WYSIWYG	What You See Is What You Get
X3D	eXtensible 3D Modeling Language
XML	eXtensible Markup Language
XP	eXtreme Programming

ACKNOWLEDGEMENTS

I would like to express my thanks to many people that contributed to this work either directly or indirectly. My gratitude goes to:

My thesis supervisor Prof. Daniel Thalmann for his guidance, support and confidences. Four years ago, I came to VRLab (it was then called LIG) for my master diploma project with rather different plans, but his lectures on computer graphics and his research captured my imagination and I decided to pursue my studies in VRLab.

The Brave Developers team: Michal Ponder, which initiated the VHD++ framework and introduced me to CBD and Xtreme Programming methods. Branislav Ulicny for sharing his view on AI management as well as the time, we spent together discussing on retro gaming. George Papagiannakis for our intensive collaboration and expeditions across Europe in the scope of EU projects. Jan Ciger for his advices on cross-platforms issues and general views on systems architectures. Ronan Boulic for his advices from academic perspectives and for his walk engine.

I am grateful to Mireille Clavier, which act as an artist and a designer at VRLab. Without her support, there would be much less “pretty pictures” throughout the document. In addition, she bravely faces the challenges to learn and use our framework (XML, scripts, HFuSM...).

Special thanks go to Etienne de Sevin, which was brave enough to become my official beta-tester for many aspects I was introducing in our framework (multitasking, scripting...).

My gratitude goes to William Damon, Graham Rhodes, and Brian Schwab for having proof-reading some chapter sections.

Naturally, I also want to thanks my family for their support. My brother Frédéric, who letting me playing with multitasking OS on his “old” Amiga 500 and gave me my first programming lessons in Basic. My gratitude goes to my parent Francis and Erica Schertenleib, who have always been supporting my choices and were very encouraging for pursuing further in my studies.

Finally, my apologize for all the people I am forgetting to mention, especially my colleagues at VRLab and Miralab (sorry folk).

This work was partially supported by the Swiss National Research Foundation and the Federal Office for Education and Science in the framework of the European projects JUST, LIFEPLUS, ERATO, NEWTON, and EPOCH.

Chapter 1

Introduction

Nowadays the society is rapidly transforming its sociological interaction with the world due to the unprecedented advances in the information and telecommunication technologies. These transformations are changing deeply the ways individuals react and interact with their environments. The variation in the ways people communicate, work, or entertain is in a constant evolution that was even not possible to imagine a couple of years from now. With the recent expansions of computer technologies, elaborating complex applications, which compute dedicated tasks efficiently, become possible.

In this thesis, we will focus on one dedicated field related to computer applications known as the 3D real-time simulations (3DRTS). Those applications create believable virtual worlds within real-time constraints. The recent progress in computer graphics technologies and powerful specialized coprocessors allow to elaborate simulations with unprecedented scenes realism and produce believable and populated worlds. This offers new challenges and degrees of complexity for creating Virtual Reality / Augmented Reality (VR/AR) systems as well as for interactive video games. Among the advances coming with faster and better hardware, the user's expectations have also grown deeply. Today, 3DRTS cannot rely only on one particular aspect, could it be graphical or its interaction model. All the different components that affect the simulation have to meet some quality, which is continuously increasing with time. Current interactive worlds feature a rich amount of diverse contents. They also feature networked abilities of seamless worlds, while populated by believable characters reacting properly to their environments. This increased complexity is raising the development budgets, which can commonly reach into the eight digits figures. This leads to improve the development process for exploiting the potential available by current technology.

1.1 Domain

Interactive simulations have grown in scale and complexity since their humble beginning in the 1960. Modern simulations have reached incredible realism and complexity. In the early ages, interactive simulations were built using primitive 2D shape geometry. The interaction models were restricted to a limited palette of movements. The beginning of the 2D era was too limited for expressing any personality and no real animations were possible. With the appearance of 8 bits computers, it became possible to create basic sprites for representing characters; even so, they were too simplified for forming any emotion. The improved technology appearing with the next iteration and 16 bits hardware offered the ability to assign to each character a set of unique movements and gestures that make them identifiable. The next leap in technology came first from adapting technology, used previously for animation movies, helping to spread real-time 3D graphics for both PCs and embedded platforms. The first 3D characters were composed by a few hundred polygons but by adding the third dimension, the characters were becoming more believable because of a bigger set of movements. Over the rapid development of 3D chipsets, the characters became more and more complex, which gave them unique personality and a limitless palette of gestures. Today hardware allows rendering close to photorealistic images. Thus, graphics improvements alone are not anymore enough to improve the simulations. For extending interactive simulations, virtual worlds should live and react more accurately. As an outcome, dedicated efforts are needed for creating realistic physical worlds and better virtual humans' behaviors. Instead of exploiting one subfield of interactive simulations, the development effort is spread in many directions simultaneously; from graphics to physics and AI. This leads to vast investments combined with intensive researches and developments in diverse fields.

Nowadays, the installed base into homes of PCs or embedded platforms for 3DRTS has reached hundred of millions of units worldwide. If once, hardware for 3DRTS were restrained to few research laboratories and government-based institutions, we are now reaching a point where a majority has access to nearly similar software and hardware technologies at a comfortable price. The implication is twofold: the actors in this medium cannot rely anymore on the hardware evolution alone, and to remain in this business, this is critical that they can compete on providing efficient solutions. The competition obliges developers to come with alternatives that can reduce the developments costs while allowing creating unprecedented experiences.

However, despite the significant advances in software engineering, many VR/AR or game engines still do not employ state of the art software engineering practices when it comes to system flexibility [Rollings2000]. The current approach is to design a custom architecture for each single project. Sometime, the system can be reused across projects, but it is usually more the results of individual experiences rather than on a formal design conception. It is usual for development houses to move from their ideas directly to coding, where success or failure depends almost entirely on the developers' skills and experiences.

So far, there is no standardized architecture that unifies the interaction between simulation subsystems and that would still allow for flexibility and expandability. The complexity to produce robust code for 3DRTS comes from the constant platform evolutions. Well-known efficient techniques used on previous iterations of hardware may not perform well on current or future platforms. This increases the chances that significant elements will be rewritten. Because of the scale of interactive simulations that are commonly composed of millions of lines of code, having to re-implement similar algorithms is a waste of resources. The time where particular interactive simulations were able to rely on the quality of a limited number of elements is over. Instead, successful systems will rely on the way all the elements interact with each other rather than on the intrinsic qualities of each element. The major reason is the combination of the content creation and exploitation within a single framework through a consistent pipeline. By offering an architecture that can evolve and adapt itself across different iterations of hardware, allowing creating simulations that do not rely too heavily on the existing platforms, hiding the internal complexity to specialized programmers. As an outcome, designers can concentrate on their ideas and the interaction mechanisms rather than focusing only on hardware limitations. The scale and complexity of future interactive simulations will distinguish the successful solutions, which will be those capable of handling the constant evolutions at a lower cost.

1.2 Motivation

The software development practices for interactive simulations are undergoing a constant evolution. The development costs are now reaching the production values of movies. However, the production models do not offer yet a similar separation of labor that would allow building applications without being too restricted with technologies or experiences. This lack of modularity comes first from the financial models where the software publishers control all the incomes. They need to run their business on a quarter basis. Therefore, the potential and risks associated with developing innovative solutions for interactive simulations remain limited. Slowly, this is changing, as current successful projects are the ones that manage to find the proper balancing between many heterogeneous functional elements. Every element from 3D graphics rendering to physics or behaviors creation requires complementary knowledge. However, combining those elements in an efficient and simple way may not be so obvious. The integration of each technology may still benefit a lot from individual researches but their exploitations need much more dedicated attention. Often, we can observe that the integration and interaction between those elements rely on a monolithic piece of software. The underlined reasons are the lack of a well-understood and standardized strategy to combine them. There is a need that the whole addition become more attractive and functional than a one-time combination. Instead of adopting a flexible framework, many existing systems tend to look for replacements and extensions, resulting in a decrease of clarity and loss of performance. If such approaches were possible when a limited number of simple entities were populating virtual environments, it will become soon too laborious to elaborate such simulations, because of the ability to produce simulations featuring hundreds of believable entities in real-time. The missing element is to adapt and extend some well-known engineering practices for massive scale systems to the 3DRTS area, notably for crowd simulations. Thus, a system that will result in a consistent and seamless unity while being controlled and configurable entirely by its data is necessary. This will also expend the potential of interaction by allowing designers to not rely entirely on the programmer's skills.

1.3 Assumptions and Challenges

In the near future, the size and complexity of 3DRTS will force developers to acquire new methods. The most visible symbol in this evolution is the technological advancements. With time, computers embed more power (highest speed, more memory...) that offers new possibilities for designers. However, these progresses have a limited impact on the simulation itself. If technology is necessary for creating believable virtual worlds, they are not a sufficient condition for creating unique 3DRTS experiences. Future 3DRTS developments will require promoting a better access to the underlined technology. By letting non-programmers interact more easily with 3DRTS, new ideas, less driven by technology, may emerge. Thus, software engineers face interesting challenges requiring re-thinking architecture designs. This may lead to promote new programming

approaches along the Object-Oriented Programming (OOP) paradigm such as graphical or non-algorithmic programming languages. This also required a change in content creation methods to enforce procedurally generated objects for creating buildings, landscapes, or virtual humans. One side effect with better 3D capabilities is that the notion of disbelief disappears and simulations aspects such as characters animations need to reflect the rendering quality with better interactions with the environment. Therefore, combining all these elements with the available technology require strong methods. In this thesis, we illustrate set of patterns for creating a flexible architecture, which target both programmers and non-programmers. This is important, as the 3DRTS industry has little time and money to invest on experimental research. In contrast, academic research offers many possibilities to explore news areas and ideas, which may in turn be used by the industry.

1.4 Approaches

The main contributions of this thesis are new approaches to define architectures for 3DRTS. In the course of this work, several new solutions and patterns were found and will be covered in this document:

- Clear separation of components that encapsulate platform specific code on low-level layers.
- Set of design patterns that targets 3DRTS for concurrent architectures and workflow management allowing to runs applications across several iterations of hardware.
- Multi-layers architecture allowing combining optimized low-level routines with high-level design.
- Concurrent programming scripting abilities that automate concurrency management. These free non-programmers from low-level technological issues. This is important, as next generation of computer hardware will feature highly multitasking designs.
- Data-Driven mechanisms to control simulation events and notably agent's behaviors by clearly separating both the logic and data layers.

The combination of these contributions illustrate that successful future architectures will not necessary being the one that can push the hardware to its limits but rather the one that can unlock its potential for all the different actors involved in the development process, including artists and designers. This thesis intend to provide a better understanding on software design as well as offering good flexibility for artists but also still continuing to give opportunities for field specialists to develop their ideas. The goal is to reduce the requirements of becoming an expert in the entire domain.

1.5 Structure of the Thesis

This thesis is subdivided into 11 chapters that can be categorized in four groups. The first group (chapters 2-3) presents the problematic and related work. We will analyze the difficulties associated to the creation of an efficient architecture for 3DRTS. The second group (chapters 4-5) covers architectural patterns. In addition, we will analyze the ability to combined high-level design with efficient real-time performance. The third group (chapters 6-8) presents concrete implementation details of our architecture from hardware-oblivious algorithms to high-level and data-driven scripting interfaces. The last group (chapters 9-10) highlights concrete exploitations of the platform using toolsets and use cases

The next sections introduce the topic of each chapter with more details:

In chapter 2, we introduce the main concepts and terms related to the design and development of a modern, large scale, data-driven system for 3DRTS. In particular, we will discuss the impact of creating massive scale applications within the development process. We will follow the technical issues for handling a team of developers and artists and contrast with the limitations that monolithic applications suffer due to improvised developments and design decisions. We will also emphasis on describing the different elements and their own constraints with a particular attention to combine all theses elements into a clear and evolutive design.

In chapter 3, we will perform an analysis of the evolution and current technical approaches use by 3DRTS developers. We will analyze the benefits and limitations of existing approaches, while also discussing the current trend in extending such platforms.

In chapter 4, we will concentrate on describing the architecture retained in this thesis, describing its strengths, but also pointing to areas that may benefit from extended analysis.

In chapter 5, we will analyze the impact and adaptation of this architecture with the current and future technology. We will demonstrate that high performance can still be achieved with high-level data-driven architectures. To illustrate our purpose, we will discuss about dispatching tasks within a multithreaded environment and its impact on the system development.

In chapter 6, we present the innovation in computer hardware and their direct impact in 3DRTS developments. We illustrate our talk by describing some low-level components developed to keep excellent performance. This will lead to understand how to take full benefits of existing technologies and we will discuss the increasing importance of concurrency in current and future hardware and their potential impacts on such critical system elements. Finally, we will highlight the interaction model between the entire system elements. A dedicated section will describe how the modules can be shared and reused for different simulations or hardware, and how the workflow can be balanced dynamically between the different coprocessors for keeping interactive frame rate.

In chapter 7, we will perform an intensive description of an innovative scripting layer that provide to designers and scripts writers the abilities to write efficient scripts within a concurrent or cooperative multitasking environment. It avoids the pitfalls that come with classical multithreading environments. This chapter will also discuss the benefits and real-time performance of this approach that extend the usability and prototyping abilities of such systems.

In chapter 8, we will introduce a dedicated module for controlling large amount of virtual character through a pluggable and data-driven AI engine. This will lead to explain the system adaptability toward different applications including the most demanding like crowds simulations.

In chapter 9, the attention will be dedicated on describing that developing a robust piece of software is no more enough for creating complex interactive simulations. Today content creation requires to be planned carefully within the system. The creation of proper authoring tool for preprocessing and real-time interactions is primordial for unleashing the system functionalities.

In chapter 10, we focus on the validation of the proposed methodology from the perspective of the different actors involve in the development process (artists, designers, programmers...). The validation will shortly present different applications and highlight the wide range of applications that the system is capable to handle. This includes augmented reality simulations, crowd simulations, and storytelling systems featuring advanced virtual character interactions and input devices.

In chapter 11, we will summarize the thesis and discuss about potential evolution and ideas. We will also analyze what the lesson learned during the research process and see how architectures for interactive applications may evolve in the future.

Chapter 2

Concepts and Terms

This chapter will introduce the concepts and terminology associated with 3DRTS architectures. It is divided into several sections that first present a basic definition of the domain. The next part provides information on the problematic of collaborative works, required by the current scale of 3DRTS developments. This has a direct impact in the choices of tools and methods for managing the developments. Most of the concepts presented in this chapter are common to any software developments, but are particularly important for 3DRTS developments.

2.1 Architecture

Systems architectures for interactive simulations are some of the most complex software being developed. They have to evolve continuously, following the technical advances in computers hardware as well as increasing the interactivity and realism from the previous simulations. If we refer to the ANSI/IEEE Std 1471-2000 definition, architecture is: *“the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principle governing its design and evolution”*. The Open Group Architecture Framework [TOGAF] is extending this definition by describing architecture as *“a formal description of system, or a detailed plan of the system at component level to guide its implementation”* and *“the structure of components, their interrelationships, and the principles and guidelines governing their design and evolution over time”*.

For end-users, the frontier between the simulation and the system that powered it is not always clear. The architecture responsibilities include strong organization, limited dependencies, and associations between modules [Booch2005]. The advantage of developing systems for multiple simulations is to allow reusing many existing components. The architecture can define all the non-specific technology into a well-structured environment. Every simulation relying on the system will provide their own assets (3D models, animations, scripts...) as well as extending the platform for dedicated purposes like controlling input devices or producing authoring tools and behavioral scripts. In the next sections, we will briefly analyze the different components that are likely to be present in any 3DRTS frameworks.

2.2 Interactive Simulation Systems

The definition of an interactive simulation system can be slightly different upon the perspectives. At least, it define the underlying software that control and execute the simulation in real-time. It provides a set of minimal functionalities. For instance, in the past, 3D game engines were considered as 3DRTS systems where the 3D renderer was the principal element. However, current simulations are providing much more information and interactivity. Thus, the current definition will rely on providing all the software elements within clear Application Programming Interfaces (APIs), while offering tools for previsualization and content creation. The interests of using systems architectures for interactive simulations go beyond the rational development costs. These allow concentrating on the content creation and interaction models. The technical designs of successful architectures for 3DRTS require a framework of reusable components, good efficiency, as well as offering to non-programmers the ability to manipulate the simulations in a safer environment. Naturally, the proclaimed goal is to ease the development process and keep the development costs as low as possible. In fact, the Component-Based Development (CBD) methodology fit perfectly theses requirements [Cho2002]. However, the actual principles and concrete implementations are still an active research field. This thesis is an attempt to provide the elements of comprehension for adapting such concepts for 3DRTS. For instance, in the OOP programming paradigm, the unit would represent actual objects [Stroustrup1991]. In CBD, architectures are defined by their components. Every component represents a stand-alone unit that minimizes the interconnections. By applying this scheme, architectures do not rely on particular modules or technology but can more easily reassembled independent components. It offers better code reusability over monolithic systems. These old fashion systems are suffering from high dependency levels where components interfere with each others [Stutz2004]. CBD differ from other technical approaches by trying to separate the software into

multilayer of independent packages. This has the drawback of increasing the communications layers as well as duplicating some information in different modules. However, the advantages are important, as the maintainability and adaptability is much higher, resulting in dedicating less time for refactoring major code base at every project. In fact, the creation of well-designed architectures for 3DRTS has emerged only recently. The increase complexity and size of current projects force the adoption of stronger software engineering practices. Moreover, being able to reuse many existing technologies provide a more streamlined process and increase the proportional time dedicated for fine-tuning simulation parameters in regards to the time spent building core components. Any architecture will come with several limitations that may reduce its usability. Basic design decisions can have important impacts that may be difficult to overcome later in the development process. In addition, developers have to follow the conceptual design reducing the freedom given to the developers. Another problematic consists to understand and master the engine and its design. Even so, programmers do not need to understand the underlined components; it still requires an important learning curve for discovering the technical limitations and abilities offered by the architecture. It obliges developers to make the effort to analyze the system. If all those disadvantages may seem discouraging at first, many of them can be reduced or removed altogether by applying appropriate software engineering methodologies.

3DRTS runs on top of an Operating System (OS) and hardware. Systems architectures provide additional layers such as APIs and modules. This multilayer representation is necessary to control the complexity and for abstracting platform specific components. The creation of modules serving as system components have to combine the OS layer with higher levels of abstraction through consistent Application Programming Interfaces (APIs). From a simulation designer perspective, systems architectures are acting in a similar way than a real OS abstracting the technical issues under low-level components, while offering set of functionalities to control and create dedicated contents. Modern architectures have to evolve dramatically to escape the notion of adopting low-level tricks as system foundations. The first attempts were always built with particular simulations in mind. For instance, some game engines were more adaptive to specific game genres such as Role Playing Game (RPG) or First-Person Shooter (FPS), while VR systems were tightly connected to specific hardware. Now, the current trend is to clearly make the distinction between the engine and the simulations built on top of them.

2.2.1 Visualization

In much software, the interaction with end-users is done mainly with texts and 2D shapes. Naturally, the principal component of 3DRTS and one of the most resources demanding is the 3D renderer. Its purpose is to display 3D images on a screen. It visualizes the scene, which provides a visual feedback to end-users. Generally, this element is so critical in the realism and sensation of immersion that much architecture constructs their system around their 3D renderer. Often, notably in video game development, this single element is using more than 50 percent of the whole CPU resources and all the GPU resources. With the trend to produce more and more realistic rendering, this component suffer from being dependent to outside vendors for both software APIs and 3D hardware chipsets. With the appearance of hardware capable of accelerating 3D images computation, the possibility of generating close to physically correct algorithms for special effects like scenes illumination have diffuse the medium to a wider audience. The 3D renderer principal responsibility is to optimize the computation of 3D geometry. As an outcome, dedicated efforts have to be planned for taking advantages of hardware particularities like memory bandwidth. One drawback with 3D renderer is that they become depreciated as soon as a new iteration of 3D hardware is available (see Figure 2.1).

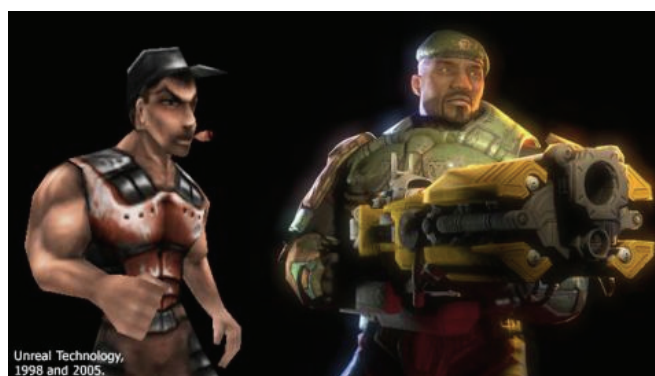


Figure 2.1: The evolution in 3D graphics showcased by the Unreal Engine.

2.2.2 Character Animations

As far as the graphic processing power is increasing and that rendering techniques become closer to their original models, character animations artifacts become more disturbing. The ability not only to render but also to animate virtual characters can be critical to the sensation of presence. The current 3D graphics have suspended the notion of disbelief. Previous approximate animations engines are not anymore sufficient. The current level in characters animations takes advantages of techniques that were only possible for off-line rendering a couple of years before. Nowadays, character animations use skeletal modeling systems (see Figure 2.2) with the potential addition of physically accurate controls using Inverse Kinematics methodology [Magenat-Thalmann2004]. Skeletal-based animations offer to control each vertex that composes the character using several bones. This allows moving body parts like the arms that in turn affect others body parts. The overall effect is a far more believable set of animations. Moreover, with the expansion of hardware capable of computing vertex shaders, skinning computations can be made entirely on the GPU [Nischt2006].

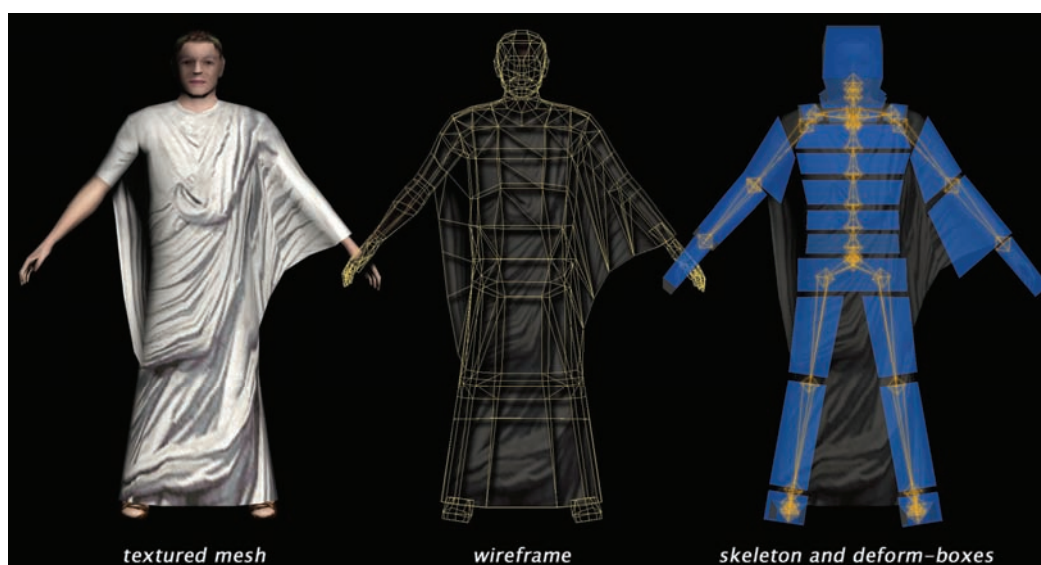


Figure 2.2: A virtual human and its skeleton.

2.2.3 Level of Detail (LOD)

Level of detail is not a new idea. In the past, they were used for reducing the geometry complexity based on the distance between the camera and the considered object. However, with the constant evolution and wide range of hardware on the market, architectures must extend the notion of LOD for several reasons. One being that geometry is not the only element that can benefit from LOD. Components like AI or sound are also good candidates to scale down their computations on demand. Developing and thinking in terms of LOD is connected to the creation and management of assets. For instance, instead of creating 3D models for the current set of hardware, it is more efficient to create high polygon models that can be scale down. This increase the adaptability to different platforms as well as benefiting from 3D techniques like normal mapping [Maughan2003, Huddy2004, Zarge2004, Green2005]. In addition, assets can be reused over different projects and maybe shared with others industries [Kane2004, Ancel2005].

2.2.4 Load-Balancing

Whatever platforms, developers are always restrained by the processing throughput available. The technical constraints are strong limitations toward designers' ideas. To overcome those problematic, 3DRTS have to minimize the computation of unnecessary data. In effect, 3DRTS may have rich environments representing large areas of space. However, at a given frame, only a small subset will be visible. By not processing those objects, the workflow can be reduced. Many culling methods for removing non-visible objects exist. In this thesis, we will discuss on a specific implementation, which take advantage of the cache memory layers for faster performance. For keeping efficient load-balancing, modern architectures should manage cache memory due to memory bandwidth bottlenecks. Therefore, architectures that can reduce memory transfers avoid

wasting important CPU cycles on Input/Output (I/O) operations. There are various techniques for keeping cache trashing to a minimal that will be discussed in details later.

2.2.5 Scripting Systems

Much current architectures provide complete or limited scripting functionalities to their system. It gives better prototyping capabilities. Scripts may have different use; they can control non-interactive cinematic using the 3D engine or to control virtual characters behaviors. Alternatively, whenever flexibility is more important than pure performance. For instance, writing behavioral scripts is likely to be more streamlined than hard coded them in C++. Scripting systems are also useful for allowing team members with limited programming knowledge to interact with the system, as scripting systems handle most of the complexity that affect C++ coders, like memory management. In this thesis, we will describe an innovative scripting interface that goes beyond the simple text-based, single-threaded style of scripting systems.

2.2.6 Artificial Intelligence (AI)

In this thesis, one scope of interests concerns the management of large amount of living entities. Crowd simulations come with their own set of specific requirements and oblige the system to be flexible for managing the entities. Ideally, each agent within the simulation should react independently and accurately with the environment. AI engines managing crowd simulations need to provide functionalities to generate AI stimulus within a complex environment. Events need to be propagated and realistic 3D navigation maps for allowing characters to move freely in the environment is necessary [Ciger2005, de Sevin2006]. This is primordial as it is not possible to assign manually tasks to each individual. The consequence is that decision routines have to be well defined. As the development process is an iterative work, the AI system should be driven by its data, as shown in a later chapter.

2.3 Middleware

The developments of systems architectures for 3DRTS require developing and integrating different elements like graphics, AI, sound or physics modules. Middleware are software that supports developing applications by exposing the hardware powers in a safer environment. They provide the common functionalities and specific requirements. The Figure 2.3 depicts the different layers involved when using middleware.

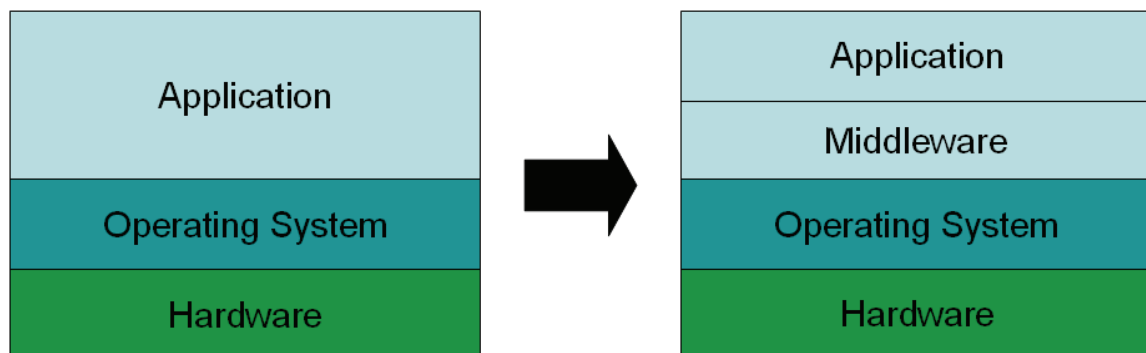


Figure 2.3: The middleware layer provides a software abstraction between the application and the platform specific operating systems and hardware.

One might argue about the advantages of relying on third-party components when developing its own applications, as they come with some financial and technical risks, whenever the middleware vendor goes out of business for instance. However, developing 3DRTS using middleware software comes with many advantages [Merceron2004, Dhupelia2005]. Current developments require a vast domain of expertise to handle the entire components. Being able to rely on technologies that come with existing validations prevent the risks involved when trying reinventing the wheel by improvised developments [Milo2005]. Developing 3DRTS from the ground up is more expensive than using middleware, as the development team need to hire many experts that

need to be trained with the latest technologies. Moreover, developing high-end technologies is a very slow and unsecured process. The team may end-up with technical solutions that do not suit anymore the market. By relying on middleware, developers can concentrate on building their applications and benefits of the technical advances provide by middleware vendors. However, it would be unfair not to point out some few disadvantages: the license costs can be important investments and integrating existing assets can be difficult. In addition, the need to come with unique features and specific customizations may not always be compatible with the use of middleware.

2.3.1 Middleware Components

The current middleware market is split into different categories. Some vendors provide dedicated modules like physics simulations [Ageia, Havok, Newton, Tokamak], 3D graphic rendering [Emergent, TouchdownEntertainment], audio [Firelight, RAD], AI [BioGraphics, Kirby], network [Frisbie, GarageGames-TNL], math [Intel_Math] and special effects [Bionathics, IDV]. The second category is middleware that provide the complete range of software and tools required for building 3DRTS [Criterion, Epic-Unreal, Virtools, Valve-Source2004]. In many situations, they are integrating some of the middleware from the first category. Middleware products include generally an Software Development Kit (SDK), composed by a library (mainly in C/C++), tools and exporters from the majors DCC software like 3DSMax [Discreet] or Maya [Alias]. Finally, they provide documentation and examples of the middleware capabilities. Since the middleware serve as layer between the application and the platform, many middleware products are cross-platforms.

2.3.2 3D APIs

For 3D real-time graphic rendering, developers rely on APIs, which present a consistent front-end to an inconsistent back-end (the 3D hardware). For instance in the PC world, two major graphical APIs coexist. The first one is the [OpenGL] API, which is a true cross-platforms solution. It is the principal choice for cross-platforms developments and within academic and CAD institutions. The ideas behind this API were to define a set of common functionalities that the different 3D vendors were trying to implement in their hardware. However, as vendors were diverging on the future of the API, dedicated extensions of unique features have been created making programming more difficult. The other contender is the Microsoft DirectX API [DirectX]. The benefit of this API is that it can evolve whenever Microsoft decides and therefore it can dictate which functionalities will become available to programmers. As DirectX become the API of choice for Windows and Xbox only titles, 3D vendors are now dedicating more efforts to fulfill the DirectX requirements than trying to improve the OpenGL API. As a note side, an initiative from Khronos with the OpenGL ES API [Khronos], which is the supported 3D API for the Sony PSP and Sony Playstation 3, may become an interesting alternative to the current APIs. Whatever, the major distinction is more the methodology chosen for exposing the 3D graphic functionalities as well as providing materials for long discussions to decide on the best 3D graphic API (see Figure 2.4).



Figure 2.4: 3D graphics developers have to choose between the OpenGL and DirectX APIs.

2.3.3 Digital Content Creation (DCC) Tools

For producing simulations assets, artists, and designers need the support of tools that are clearly oriented toward the content creation. Artists need to be able to work in full feature and mature environments that are not restricted to particular engines or APIs. These tools provide set of functionalities for creating complex and original assets. With them, artists can concentrate their energy on the content creation process rather than dealing with time-consuming technical problems. For programmers and managers, the advantages to have the artists more independent of the technology are twofold; artists can already begin to work before the software technology is built and the interdependence between programmers and artists is considerably reduced. Programmers can restraint their support for creating exporters and plug-ins that fit the system architecture requirements. Theses exporters will generate either binary or meta-data files that can be manipulated outside the DCC tools. DCC tools can be categorized as levels and worlds editors, 3D modeling software or plug-ins for others software such as 2D imagery software [Adobe, Corel], sound mixers [DirectX_Dev_Team] or shaders editors such as FX Composer [Maughan2006] and RenderMonkey [ATI-RM]. In the next sections, we will briefly provide an overview of these DCC tools. The section 9.1 describes the ones use effectively in our system in more details, such as the open editor for OpenSceneGraph [osgEdit] or the oFusion editor for Ogre3D [Carrera]. With the help of those tools, artists get immediate feedbacks on the content creation and can analyze the performance impact of different contents (shaders, polygon count...).

2.3.4 Environments Editors

This category of tools is built on top of specific frameworks. In the past, these tools were use internally by designers for creating simulations or game levels. With time, some of these tools become available for the mod community. They let homebrew developers extending the functionalities of existing simulations. In fact, many current games and simulations offer the abilities to create new contents. The advantages of dedicating time for developing such tools are subtle. Generally, the lifetime of 3DRTS is limited to few months. Allowing homebrew teams to develop new contents for the simulations make them more popular, which increase their longevity [Hodgson2004]. The impact on product sales is also not negligible [MacLellan2000, Kushner2003, BBC2005]. Environments editors combine different contents altogether such as static and dynamic objects, as well as placing simulations events, lights or bots. Such editors are not designed to replace existing 3D modeling tools but provide lightweight versions, which expose specific engine abilities in an easier environment. Some environments editors include Hammer [Valve-Hammer] from Valve software, which is used to build environments for the Source Engine [Valve-Source2004], but also provide functionalities helping controlling and creating virtual characters with accurate AI and lips movement synchronizations. Other tools such as Q3Radiant [id-Q3Radiant] developed by idSoftware for the Quake 3 engine is widely use by both the mod community and indirectly by many open-source 3D engines for loading contents [Crystal_Space, Gebhardt, Streeting]. Sometime, they rely on the Quark editor (Quake Army Knife, see Figure 2.5) [Rigo]. Another environment editor is UnrealEd [Epic-UnrealEd] developed by Epic Games for previous iterations of their Unreal Engine [Epic-Unreal]. The UnrealEd editor provides similar functionalities for designing in-game environments. The second set of environments editors are dedicated for the generation of terrains or plants. Tools like [Demeter, PlanetSide, Soconne, Sugimoto] use several techniques such as fractals based algorithms for creating complex 3D terrains. They comes with optimizations techniques for placing 3D objects on the generated terrains as well as exposing 3D terrains data for producing navigation graphs for instance.

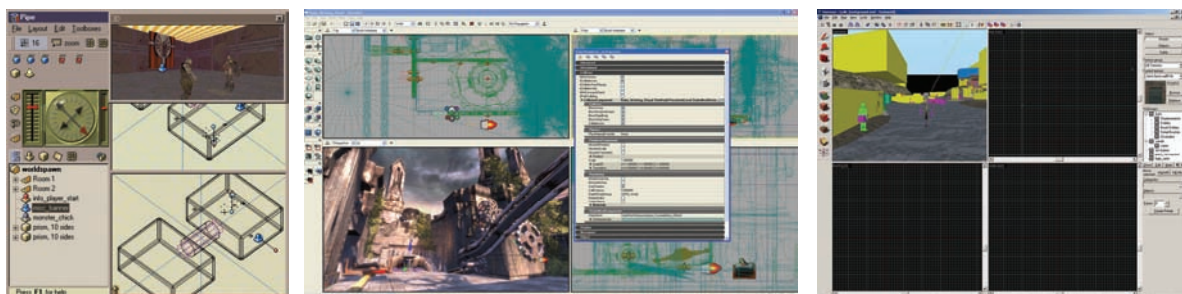


Figure 2.5: In the left, the Quark editor. In the middle the UnrealEd editor and in the right the Hammer editor.

2.3.5 3D Modeling Software

Before trying to integrate assets into the simulation, artists need to create them with software referred as 3D modeling software. The content creation pipeline generally starts with such software. The resulting assets will then be exported into engine specific files formats for real-time usage. 3D modeling tools are used for creating 3D graphical objects such as buildings, characters, and vehicles. They offer better flexibility than environments editors. They also allow creating objects that are more detailed. The most popular 3D modeling tools include 3DSMax [Discreet], XSI [Softimage], LightWave [NewTek] and Maya [Alias]. Every software solution has advantages and disadvantages to perform special tasks. Some 3D modeling packages facilitate characters animations, while others will be better for creating special 3D effects. Generally, artists need to master several applications for creating all the variety required by current applications standards. The Figure 2.6 depicts the usual look and feel of 3D modeling packages.

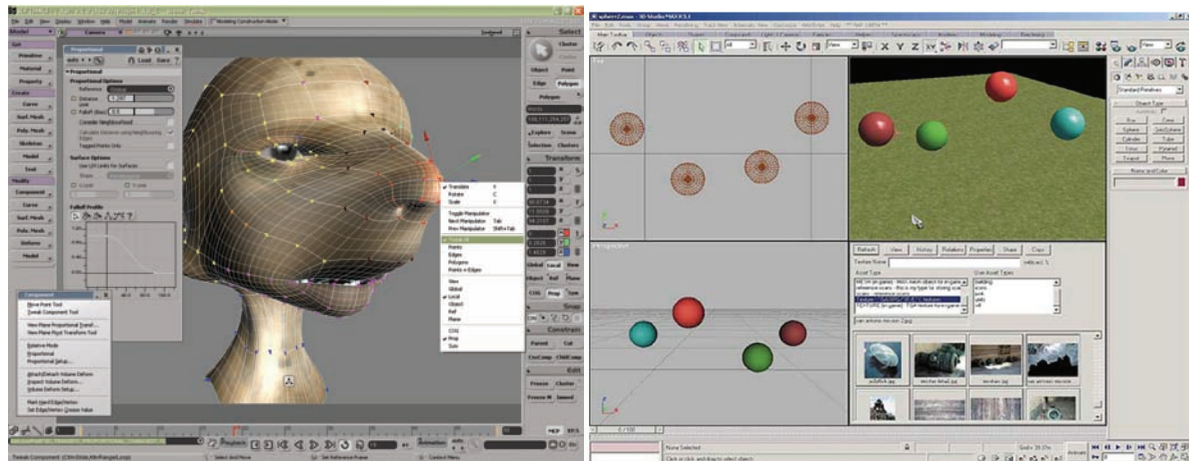


Figure 2.6: 3D Modeling tools allow creating 3D geometry within a dedicated environment (on the left: Softimage, on the right: Maya).

Because of the high-level of competition, 3D modeling packages constantly integrate extensions as standard features. This obliges plug-ins companies to continuously extend the software capabilities. However, developers still need to elaborate custom plug-ins and exporters for loading and saving assets into specific files formats. In addition, many simulations need to extend the functionalities for fitting specific needs, could they be to provide unique features or to facilitate the content creation pipeline. As an outcome, all 3D modeling packages come with a SDK that make the creation and exposure of their software functionalities available through plug-ins. The communication can be done either using APIs such as the Maya API [Alias2005] or MAX API [Murdock2005] or directly by developing scripts with customized languages such as MelScript [Gould2002].

2.4 Methodologies for Systems Architectures

Strong software engineering skills are necessarily for producing more predictable and faster results, while reducing the problematic of on the fly implementations. In effect, interactive simulations have reached a scale that require a good methodology to keep in track the development process [Bethke2003, Demachy2003, Irish2005, Krueger2006]. It is common to see teams from 25 to 100 developers working on the same project for a period of 2+ years. Thus, it becomes more and more important to adapt and improve well-known software engineering practices for 3DRTS. This adaptation means a change of mentality, moving from projects, where developments consist of unstructured and iterative processes [Booch2006, McConnel2006]. Certainly, the constant novelty may not always work well with some software engineering methods relying on highly structured and predefined methods and their associated overheads. However, not all software engineering approaches are following those ideologies and 3DRTS systems need naturally to adapt the more agile methods. The Figure 2.7 describes the hierarchical layers involved during the development process relying on the Agile development [Keith2006]. The intrinsic characteristics are to enforce people and communications over processes and project automations. This method increases the flexibility when responding to change.

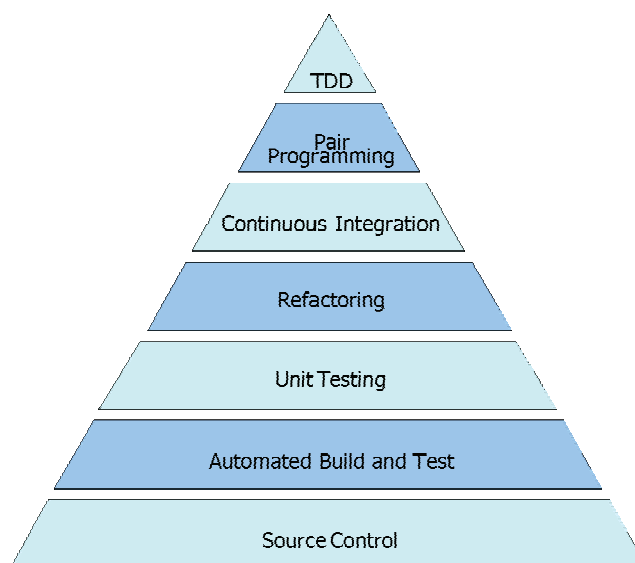


Figure 2.7: Pyramid of the development process methodologies [Keith2006].

2.4.1 Technical Design

To conceive massive scale software architectures, they need to be organized cautiously by going through the process of writing technical design documents. Their purpose is to analyze the architecture, its goals, and its limitations before thinking to the purely technical issues. This stage within the development process is critical, as the wrong decisions will have an impact over the whole system lifetime. The technical design can be a combination of exchanging ideas with co-workers or by applying methods like UML for both a communication and starting point tool [Douglass1999, Jacobson1999]. Or more accurately by creating schemas within UML designer tools such as Rational Rose [Rational] or UML Pad [Bignami]. In addition more flexible and agile development process such as the Scrum can fit the specific requirements of 3DRTS developments [Keith, Scrum]. With Scrum, projects progress via a series of month-long iterations called sprints. From a different perspective, [Foltz2003] describe techniques to ease code maintenance. Using these methods, we can greatly improve the development process.

2.4.2 Collaborative Work

For historical reasons, past 3DRTS developments were mostly the results of homebrew or small teams working close from each other's to prevent the need to share information among team members. However, with larger teams and important shared resources, the question of knowing how to share that information and keep it up-to-date becomes critical. Collaborative work between programmers is a difficult task. Every actor will have its own coding style and methods that goes beyond the look and feel of the code. A good approach is to emphasis naming conventions and coding styles over the project. However, the different styles between the developers will still be visible. To reduce theses side effects, one solution is to clearly separate responsibilities among team members. However, this may lead that components will fork or that a developer will be faced to a problematic that another team member would be able to resolve faster. Even if the development progress do not suffer from theses situations, this is also dangerous as the knowledge on large amount of code base can be lost if the maintainer leave. One solution is to apply code reviews between a limited numbers of people, which ideally associate a senior developer with newcomers. The advantages are twofold and consist of improving the quality in code and in a way to spread the knowledge about the design and the code base. To keep the team coordinated, the information about the progress made in the different components and their implications have to be clearly disseminated. Some specific ways of sharing the information include regular meetings, internal websites, and mailing lists. Also, the use of Wiki [Leuf2001] or Mantis [Mantis_Dev_Team] websites allow any team member to annotate running issues or technical advices, are some of the tools to consider when developing massive scale software. However, any of these tools still rely on the actors and their good willing of checking and keeping the information up-to-date.

2.4.3 Code Maintainer

3DRTS are composed of many components from low-level libraries to high-level modules and user interfaces. Over the time, some team members will leave or integrate the project. Thus, the existing code base will have to be maintained. Different approaches are possible: the *fairly strong code ownership*, *consulting ownership* or *free code ownership*. Here, we are not talking about the legal owner of the source code, which usually is the institution or corporation handling the financial resources. The *fairly strong code ownership* consists of designing one team member as a unique code maintainer for a specific module. The benefit is that the maintainer knows exactly the status and functionalities of the modules but in the other hand, there is a huge potential of loosing the knowledge or being stuck if the developer is not present. The *consulting ownership* is very similar to the precedent except, it gives to additional developers the right to propose and modify the source code, whenever the maintainer cannot do it directly or immediately. This approach helps in sharing the knowledge as more than one actor know the inner code. However, it is primordial that the consultant and the maintainer communicate effectively. Finally, the *free code ownership* is allowing all the team to modify the code base as needed. As everybody can modify files and improve the overall system, the knowledge is spread among the whole team. However, this approach may not be adequate with specialized knowledge as developers may try to resolve and fix particular cases, while breaking the whole system at the same time. The second methodology is generally the best suited for 3DRTS, which rely on highly specialized code.

2.4.4 Conventions

To ensure the livability of massive scale applications, a dedicated attention has to be made on defining a common terminology among the team members. Programmers may use coding standards that greatly simplify code reviews and code reusability, while artists may define conventions on the data creation, like texture formats and size. Those standards can be of a great help when bringing new members. Different tools may help to enforce those rules like the one from Artistic Style [Artistic-Style], which automates the code formatting to the chosen standard.

2.4.5 Assets Management

The increase scale of 3DRTS requires managing all the simulation assets. Different software have been developed, which help to keep all the assets within a common repository. If such tools exists for sharing source code such as [CVS] or [SVN]. Most of them lack appropriate front-end for non-programmers and generally they do not offer integration APIs even for simple workflows. Assets Managers software is piece of technology that allow to control large-scale database of assets with APIs that allow extending their functionalities for specific needs. Solutions such as Alienbrain [Avid-Alienbrain], Perforce [Perforce], Serena [Serena] or XNA [Microsoft-XNA] improve the development management, by reducing the problematic involved with constant modifications with simulation assets. Assets management systems help for storing the information but do not prevent some development issues related to software versioning. For instance, it may be difficult to regenerate all the simulation assets from the original version stores in the assets manager. This is particularly true for systems relying on metadata files, which differ from their run-time version. Modifications affecting the assets and their generation may be distant in time. Off-the-shelf assets management solutions are tools that still rely on being used properly by developers.

2.4.6 Source Control

Software development is an iterative process, which oblige developers to control the versioning of their system. The most notorious solution includes the concurrent versions system (CVS), which is a network transparent Open Source Version Control System. This tool targets both small and large distributed development teams connected to the same network. CVS like most sources management systems helps developers by tracking and storing all the history version of files into repository and simplify the comparison with older versions. It also keeps a trace of all files even the one that was deleted, allowing retrieving files. CVS also come with merging capacities helping the developer to integrate their changes to the source code on the same final resulting files. CVS also provide code replication allowing several projects to share the same common set of functionalities, enforcing the benefit of using the same library without worrying of having to coordinate their code at every update. Developers are free to use a stable version for some period and then to benefit from merging operations for updating their software with latest available versions. The design of CVS provides a client/server system with a common central source repository, where each client works with on their own local repositories. Client

users have to commit changes they want to reflect to the central code base, but also, they have the possibility to get updated code from the server, whenever they want to upgrade components. CVS will eventually cause conflict files when both the client and server files were heavily modified. Another alternative for control system is Subversion. Subversion relies on the principle introduced by CVS but try to overcome some of its limitations. Apart from those two majors open source initiatives, a series of commercial source control tools are available such as Microsoft Source Safe [Microsoft-SS], Perforce or Alienbrain. Most of theses tools provide similar functionalities and the main advantages of commercial products is their direct integration into developments IDEs and their ability to handle binary and assets more accurately than free open-source initiatives. If open-source and research community use mainly CVS or Subversion, the trend is not the same within the industry. During a dedicated panel of developers at GDC 2004, some records were made. They indicated that 50% of the audience was using the Visual Source Safe source control system due to its integration to Microsoft Visual Studio IDE, and another 25% were using Perforce. More interesting, less than 15% were using open-source initiatives. The interesting elements of this report was that most interactive simulations developers use source control systems more for keeping an history of files than taking full advantages of source control systems such as branching or private access (see Appendix I).

2.4.7 Automate Build System

With systems composed of 50+ libraries, the problematic of keeping the code base in valid states is appearing. Generally, default projects management coming with standard IDEs do not handle well builds management and are platform dependent. The advantages of using automate build system are twofold: developers get immediate feedbacks when some modifications break their code and the code consistency is enforced. The different categories of build systems include [Perforce-Jam], [Apache-Ant] or [CMake]. Another alternative trying to combine assets and build system in one common infrastructure is XNA Build [MacMahon2006]. Being part of the XNA Studio [Keller2006], XNA Build intend to improve the builds management with dedicated tools. In our system, we choose to rely on [Scons], which ease the management of build system, by offering scripting capabilities for creating archives of source code or multiple configurations such as debug or release builds.

2.4.8 Unit-Testing Framework

The particular nature of 3DRTS has prevented the large usage of unit-testing frameworks. In effect, graphical bugs may be difficult to analyze in programmatic and automate ways. However, large amount of code such as simulation logic, math or physics routines can benefit from Test-Driven Development (TDD) [Beck2002, Astels2003]. The purpose of this method is to constantly write and run many small tests validating the input and output of functionalities. 3DRTS systems architects are investigating and integrating TDD in their architecture [Llopis-TDD2004, Schultz2005, Madden2006]. Notably, [Llopis-GDC2006] presents techniques for embedding graphics calls with TDD. For C++ programmers the principal unit-testing frameworks include [Boost-Test-Library, CPPUnit, CXXTest, UnitTest++].

2.5 Conclusion

This chapter introduced the tools and methods required for handling and building systems architectures for 3DRTS. The next chapters will introduce the current state-of-the-art in this field as well as introducing the architecture styles retained in this thesis.

Chapter 3

Related Work

Nowadays, 3DRTS have grown considerably in scale and complexity since their humble beginnings in the 1960's. However, despite significant advances in software engineering, many 3DRTS engines still do not employ state-of-the-art software engineering practices when it comes to system flexibility [Rollings2000]. A base architecture that unifies the interactions between 3DRTS subsystems and that still allows for flexibility and expandability can ease the development of robust code. The next-generation of dedicated hardware, high-end PCs and home consoles, will provide a multi-core architecture [Deloura2005, Intel2005, Microsoft2005]. Developing efficient code within a multi-core environment is a complex task that few developers will be able to master. To allow developers to create and manipulate 3DRTS on multi-core hardware, a dedicated effort is required for providing friendlier development environments. Therefore, 3DRTS systems developments need to provide much better abstraction layers for high-level interactions such as behavioral scripts. To address some elements of this problem, the component-based system approach [Adolph2004, Ponder2004] allows managing massive scale applications. This gives the opportunity for developers to focus on their specialty and take advantage of technology advances, while providing them to more people with easy access to the technology. While CBD can improve the quality and reduce development time [Alves2004], it comes with some drawbacks. For instance, Interactive Simulation Objects (ISOs) are responsible for their own data manipulation, including graphics, AI, sound, etc. As a result, CBD systems can become extremely large and complex, requiring developers to have comprehensive knowledge about the different components and their interactions with the ISOs.

3.1 System Architecture

With the introduction of the Sony Playstation in 1994, the audiences that can interact and use 3DRTS have reached a few hundred millions of homes. However, the developments of VR and games applications have been first elaborated with the very first computer rendering 2D images in real-time [Firestone2005]. In 1958, Willy Higginbotham develops one of the first interactive applications called "Tennis for Two" on an analog computer connected to an oscilloscope. It was one of the first initiatives to combine computer and screen display for edutainment and entertainment applications. However, the game Spacewar from Steve Russel developed at MIT in 1961 is generally considered the first video games in history. The next milestone in introducing real-time applications to a wider audience was the game Pong developed by Allan Alcorn in 1972. Three years after its creation, end-users were able to play the game in their home. During the eighties, the development of 2D applications allowed to identify distinct characters on the screen. In the nineties, expensive workstations like the one introduced by SGI and later with the apparition of 3D hardware accelerating 3D images generation for both PC and embedded platforms bring interactive simulations to a new level. The first iteration of 3DRTS introduced new challenges. The applications have to control camera and collision detection in full 3D environments. The character animations had to evolve from collections of sprites to more streamlined and generated animation techniques as shown in Figure 3.1. For more than one decade, the constant evolution of processing power and 3D hardware technology had the drawback that interactive simulations were relying mostly on their rendering pipeline. This trend is slowly beginning to change as current hardware can render near to photorealistic characters and as an outcome, the user expectations for more interactive worlds are now emerging [Rubin2003, Adams2004, Adams-FP2005, Hein2005]. The development of systems architectures for 3DRTS has to offer the flexibility to adapt itself to different interaction models and to evolve over several iterations of hardware.



Figure 3.1: The quality improvement of interactive simulations over the time (here a selection of some major video games).

3.1.1 Software Engineering for Interactive Simulations

Even with the complexity of current software developments related to mission-critical software such as 3DRTS, still many modern development teams continue to rely on the old principle, which specify that developers can move directly from simulation ideas to coding [Plummer2004]. This means that the success or failure depends almost entirely on developers' skills and experiences. The need to adopt stronger approaches for 3DRTS developments is necessary. Every single software development comes with its own set of requirements, but they all share a set of common methodologies. Software engineering can be seen as a systematic approach to analyze, design, implement, and maintain software. In many situations, engineers can rely on techniques such as CASE tools [Design] and other related software design methodologies [Vienneau1995]. The fundamental elements are the systematic approaches and methods used when building software. It consists to build software in a structured way by adopting predefined methods. Unfortunately, many programmers dismiss the need of adopting such methods for building software. They claim that the efforts involved in their elaborations and their restrictions are not in ad equation with programming processes. Programming single algorithm require skills that can be learned relatively easily. Thus, if software developments consist of conceives many algorithms, one might argue about the reasons to adopt systematic process for building software. The answers are numerous. As long as a programmer can develop software on its own (with its own coding style and methodology), and do not required to share his work with groups of individual, software design may not be so critical. The single programmer is free to refactor his code whenever it needs to do it, and he does not need to worry about breaking code of other. However, rapidly, the increase complexity in both code base and team management obliges developers to adopt tools and processes supporting software developments on a bigger scale. Realistic software systems that intended to be used by external developers often require more strict procedures. Software engineering can be analyzed as the collection of good methods for developing large software systems in a systematic, deterministic, and robust way. The appropriate usage of software engineering methodologies is what separates the professionals from beginners. Many authors have work on software architecture principles. [Bass2003] have provided a good introduction on software architectures by describing many concepts such as architectural methodologies and reference models. They showed the different attributes that compose software architectures and propose a set of styles that are best suited for each attribute.

3.1.2 Architecture Design Methodology

In the opposite of dedicated algorithms or techniques for graphics [Eberly2000, Watt2000, Eberly2004], physics [Bourg2001, Eberly2003, Erleben2005, Palmer2005] or data structures [Penton2003], the literature on developing VR/AR or game systems is much more limited. While those books cover the technical details on low-level APIs usage, they provide limited information on systems architectures. This microscopic approach to deal with software developments for interactive simulations is not satisfactory for massive scale software developments. The reasons of the lack of literature on architecture design for interactive simulations are twofold. The construction of many 3DRTS systems rely more on improvised solutions than on well-defined architectures design, and the great competition keep successful approaches as corporation secrets. As the development of 3DRTS become more and more sophisticated, software architectures are also increasing in complexity, to scope with large teams and code base. The need for better methodologies and better software and tools is emerging [Hannibal2000, Dalmau2002, Llopis2002, Clinton2003, Dalmau2003, Flood2003, Llopis2003, Llopis-Web2004, Chapling2005, McShaffry2005, Rabin2005]. As a result, we can see that both academic and the industry are slowly trying to come with solutions, some of them being available to the public.

One of the first initiatives from [Rollings2000] was covering existing game architectures trying to extract the possible logic of different projects. They describe the different components involved (rendering, sound, AI...) in such systems. The book tries to introduce the notion of software engineering practices for interactive simulations and describe the reasons for which those techniques were not really considered in this field. They attribute the lack of strong software engineering practices to the attitudes affecting many developers and researchers in this field. The fact that many developments results from small teams and limited project's size did not encourage embracing those techniques or the use of third-party components. The authors provide good historical information on the interactive simulations development process. However, they failed to design a concrete architecture that would provide the functionalities interconnecting all theses components together. Nevertheless, they describe that those interactive simulation architectures have much interdependences due to the general object-centric approach. The object-centric approach is a natural representation for interactive objects having their own behaviors. However, the high-level of interconnections makes this approach less efficient in concurrent environments as well as reducing the potential of swapping easily one component by another one. Around the same period, [Boer-OOP2000] illustrate how OOP paradigm can be applied as a design technique for 3DRTS. Later, [Teixera de Sousa2002] has presented a guideline for creating game systems. Even so, the book focus on beginners, it was one of the first attempts to provide information for creating real-time simulations architectures. The book covers the basics functionalities including window management, data structures but restraint itself to 2D applications.

[Haller2002] describe a generic framework and architecture design for 3DRTS. They analyze how objects communicate within the system. They explain that much architecture employ an inter-objects communications by directly calling methods of objects. This approach ensures fast communications. However, they pointed that C++ do not provide reflection mechanisms [Stroustrup1997]. This obliges the creation of meta interfaces for decoupling objects [Filion2005]. Haller et al. analyze the approach use in the QT GUI library [Trolltech], which use the signal and slot patterns for inter-objects communications. Their approach was inspired by these patterns. It consists to connect components using slot-in and slot-out, together with the reference of emitting and receiving components. This implies the creation of a state interface. They specify that adopting string as slot identifier may decrease the performance but increase the code readability, and depending on the application, this may worth the trade-off. They also describe how ISOs can be deployed using the property pattern allowing the dynamic creation of ISOs using components writer and reader.

[Simpson2002] did a complete review on anatomy of 3DRTS system. He interests was not relying on the concrete architecture but more to exert the development process. He describes the tools and problematic that affects 3DRTS developments with an emphasis on rendering optimizations such as culling methods or textures compressions. Nevertheless, his review gives an overview for both software development and content creation including a comparison between licensing an engine and creating a new one with their specific strong and weak aspects.

[Rucker2002] was attempting to address the design of higher-level architectures for 3DRTS. In his book, Rucker intends to adapt software engineering techniques for 3DRTS, by showing that a single framework can be used for creating different simulations. Rucker describes how design patterns can be applied for mission-critical software and why reuse is becoming important. He also focuses on trying to distinguish the data from logic layers for better system flexibility. However, the large overview of techniques confuses readers on accurate solutions and the described framework is suffering from high dependency graphs, which differ from his architectural model. On the same level, [Brownlow2004] presents a series of "golden rules" for creating

3DRTS. Every rule represents a simple principle and its integration in an overall architecture. Brownlow covers a variety of topics from embracing C++ and scripting to resources pipeline, Finite State Machine (FSM), and optimizations. The goal is to free up programmers time to concentrate on managing the simulation rather than on designing the underlined system. This leads to an interesting perspective pointed by [Gold2004]. He claims that interactive simulations developments still differ from normal software developments in several ways: the development is an open-ended design process and the number of control over datasets is unprecedented. As the principal goal is to provide an interesting interactive experience, the simulation had to be empirically adapted to the targeted audience. Gold also analyze that the complexity of controlling artistic content in real-time is radically different from other software applications.

[Doherty2003] showcases an architecture designed to maximize reusability in both single player and networked multiplayer modes. The architecture separates generic components from simulation-specific components by positioning the objects system to minimize data coupling. Their architecture feature a high-level representation containing the functional components, the simulation engine and data manager that interact directly with the ISOs system. Each of these modules is responsible for specific tasks. For instance, the engine handle the graphics, audio and inputs. The objects manager update ISOs' states and the data manager retrieve content from files system and other persistent storage. In their design, they do not decouple the core functionalities and the engine layer is tightly couple with all low-level modules. From this perspective, the system lacks the flexibility that CDB may offers. Doherty argumentation relies on minimizing the data flow between modules. He describes that using the objects system, as the central interface between the top-level components is primordial to defer the identification of specific data. Nevertheless, Doherty embraces the separation of the logic and data layers, which help to make his architecture more generic. Doherty also addresses the problematic of synchronizing ISO copies to improve the individual workflow of each component, which is essential for networked architecture. One interesting problem noted by Doherty concern the impact of objects systems on all the other components and compare the trade-offs between object-centric and component centric storage. However, Doherty's architecture does not address scripting languages or late-binding mechanisms for increasing the system flexibility.

[Bogojevic2003] present architecture for MMO simulations. They describe that such architecture are often complex and that there is not any single best or standard solution. They approach the problem, by layered their architecture. The first layer consists of clients who request different actions from the Logic Server segment. The clients can be seen as input to a MMOG. Whenever, a client performs an action, it is queue and handles by an Action & Response Multiplexor segments, which is located in the second layer. They introduce a third layer containing the simulation logic and data. They describe that the massive and persistent environment of MMO force the architecture to store content in database. Finally, they promote a fourth layer, or front-end interfaces serving for account management. They describe that the layers is becoming standardized as more and more games are using the Web services to deliver information such as customer service tools, consisting of client code.

In his thesis, [Plummer2004] describe an architecture for computer games as a *System of Systems*. His approach follow a data centered framework where each independent component collaborate on a central data store. Plummer promotes code reusability, expandability, and maintainability traits. His architecture takes advantage of CBD to reduce the loose coupling of components' objects. He describes the classical approach for 3DRTS development and their shortcomings such as tightly coupled and low maintainability traits. In a similar way that in our system, Plummer approach was to identify candidate architecture styles defining "*High Level Objective and Goals*". One interesting observation is the specificity of each simulation gender and how a generic solution can promote reusability. Notably, Plummer investigates the communication between sub-systems to understand the sequence diagram and system workflow. The ideas is to layered the system into self-contained modules controlled by data following the concepts promulgated by Bass [Bass2003]. Plummer focus on describing components sequence as well s providing a vertical implementation of modules validating his architecture. He also makes the same conclusion than ours by specifying that the scope and complexity prevented implementing a complete system, which features efficient real-time performance.

[Ponder2004] presents a complete VR/AR framework relying on CBD methodologies. His approach was to clearly identify and characterize the different layers defining 3DRTS. The framework present mechanisms for loading dynamic content using the property pattern and component update are controlled by a micro-kernel. In his work, Ponder re-implement many concepts such as list, vector or queue. One might argue about the underlined reasons to not endorsing the STL library, but the decisive factor was that at the beginning of his work, the STL library was not yet endorsed by the ANSI/C++ committee. In effect, his architecture leads to adopt stronger engineering practices. This thesis relies on the basements proposed by Ponder and expands the framework capabilities with accurate late binding mechanisms and multitasking data-driven capabilities.

[Goodwin2005] describe the challenges face for developing 3DRTS that run simultaneously on different platforms. Goodwin gives advices on proper development strategies for cross-platforms developments, notably the plethora of related problems. He also explains why current standard in this field are not yet mature and covers the nuance between compilers, debuggers, and OS. Notably, he also demonstrates how multi-layer systems architectures can produce code that will run identically on different machines, despite the platform making use of the same APIs. This problematic will extend with next-generation platforms. For instance, [Reimer2005] describes the growing developments costs will force developers to support most existing platforms. Without dedicated methodologies, this may have an impact on the overall quality by adapting the application to a “*lowest common denominator*”.

[El Rhalibi2005] explore the ideas that future hardware will no longer feature single CPU, but rather multiprocessor units. They describe that this require a paradigm change for conceiving 3DRTS architecture by integrating concurrency design. They explore past research in the field of parallel computing and analyze common principles that can be applied for 3DRTS developments. Notably, they illustrate standard concepts such as the Flynn’s classification, task-dependency graphs, dependency analysis, and Bernstein’s conditions. They propose a new model for managing system workflow moving from the standard sequential simulation loop. They agreed that modern architectures should go beyond simple concurrent I/O routines, math functions computed on a coprocessor or a 3D scene rendered by a GPU. In their work, they concentrate on analysing the complexity of parallel computing and propose methods to validate them in the context of 3DRTS. The structure includes decomposing the system into granular tasks in a cyclic-tasks-dependency graph, which features synchronisation primitives, thread creation objects, and a scheduler for tasks execution.

[Finney2005] approach the problematic by using the Torque Engine [GarageGames-Torque]. He intends to teach advanced concept for experienced 3D game programmers by presenting the technical choices of the Torque Engine. Finney reveals techniques and mechanisms involve in the development process using a well-tested architecture. He illustrates how developers can turn their ideas into reality. The book covers all the aspects of 3DRTS developments with a particular focus on 3D rendering. From this regards, Finney adopt the graphic-centric approach where components are directly plug into the 3D renderer. Nevertheless, by relying on a well-proofed engine validate his assertions and concepts.

[De Margheriti2005] illustrate the [BigWorld] architecture. It consists of tools, which integrate packages of high performance applications designed for creating MMOG. He explain the different layers including the BigWorld Server acting as a dynamically reconfigurable server, a client engine featuring advanced 3D graphics as well as a set of content tools creation (World Editor, Particle Editor, Model Editor). The nature of MMO simulation requires the addition of specific tools for live management and support. De Margheriti illustrates the importance of customization by embedding the interpreted scripting language Python, and by promoting techniques for content scalability and load balancing solutions. This is particularly important for MMO as the client hardware may differ greatly. The architecture offers the ability to parameters many variables including frequency updates and events using LODs. He insists on clearly separating both the logic and data layers by keeping most simulations content management in Python. The system workflow is controlled dynamically through a cell hierarchy that concentrates servers’ resources at high-density areas.

[Flynt2005] describe 3DRTS developments from a software engineering perspective. They cover all the development phases from the design up to the maintenance, by illustrating their concepts with a game serving as a use-case. The book analyzes the necessity to conceive good requirements specifications, which help to categorize priorities. Notably, they focus on methods such as UML and their adaptation for 3DRTS architects. They present a series of rules identifying design elements with their intrinsic attributes (functions, relationship, collaboration, responsibilities...). They illustrate that their methodology offer opportunities to establish some key principles such as avoiding components coupling and reduce redundancies. They remark that a bottom-up design is practical for reusing existing and proven C++ libraries. One important discussion reported by Flynt et al. concern the high-level designs and the OOP paradigm. They analyze that the specificity of 3DRTS developments may require being pragmatic in some circumstances, where OOP overheads may not be acceptable. In addition, the book covers in detail how OOP can be adopted for 3DRTS architectures by illustrating the benefits of code refactoring. They also endorse the use of design patterns for increasing the code clarity. Finally, they highlight how to create and interpret a collection of metrics to discover the system evolution over the time. These metrics gives valuable information such as the repartition of SLOC among the modules, which may indicate the quality of the design. They also collected activities on a source repository for analyzing the repartition of responsibilities.

An alternative and interesting resources center is the website maintains by Llopis [Llopis-GW]. The website contains a selection of articles, which target an engineering look toward 3DRTS developments process. The articles do not describe a complete architecture but rather give advices covering software engineering methods, project management, and tools.

Another trend is to borrow ideas from the aspect-oriented software methodology. [Bilas2002] illustrates the emerging importance of ISOs and the direct consequence of the C++ limitations to handle deep decomposition of classes. He pointed that such system decomposition does not scale well with the constant change required by 3DRTS developments. He discuss about the need to move beyond C++ to add more flexibility in designing such ISOs systems such as adopting the component model by clearly separating the static and dynamic contents. On the same field, [Duran2003] presents the difficulties to design a ISOs system. He describes common pitfalls and proposes some solutions based on it own experience. Because most of its work was primary developed in C++, which does not have fully support for the component-centric model, Duran was faced to reinvent many common object-oriented concepts such as inheritance and delegation.

More recently, [Church2004] presents a set of methods for attaching data to objects and connecting behaviors. The goal is to move from hard-coded constants to more dynamic and flexible ISOs managements. He describes particular approaches used in shipped games. He also demonstrates the compromise made between optimizations and flexibility by showcasing advantages and limitations for each approach. Centralizing systems architectures around basic ISOs that drive the simulation is sometimes referred as soft architecture, as the behaviors is not directly part of the architecture, which reduce objects couplings.

If once 3DRTS developers were not interested by more high-level architectures such as OOP or CBD, this is fortunately slowly beginning to change. For instance, [Rene2005] describe ISOs management through CBD, by describing set of patterns fulfilling these requirements. On his side, [Buchanan2005] describe how to write a generic components library that handle ISOs using design patterns.

The future of 3DRTS developments will not be about trying to reinventing the wheel at every new generation but rather to improve and built on top of existing platforms. Such a system relies on the ability to combine all the aspects involved with 3DTS from 3D rendering to physics and AI. In the past, most developers were creating their own systems to keep a complete control on their work. Nevertheless, today, the trend is more to acquired licenses from existing frameworks. Over the years, systems architectures have evolved. Following the hardware improvements from a processing throughput and memory perspectives, the inner complexity and difficulty to maintain and built such systems, have obliged the developers to face theses new challenges. The choices available to the simulation designers have extend the variety in genre toward new directions. The current hardware can render impressive and realistic environments in real-time. Thus, the user's expectations are concentrating on other elements of the simulations [Bogost2006]. The AI and animation engine have to match their graphical representations. Technology that will deliver simpler tools for developing such environments is necessary.

The technology involved with 3DRTS systems combine many components as describe in Table 3.1. Depending on the simulations requirements, some of these elements may become optional. Combining those entire elements, which differ in capabilities and design is a huge task that requires large amount of time and resources. Commercial products have also to provide good interactions layers with DCC tools and provide the ability to be customized for specific needs. The existing systems architectures can be cataloged in different groups. The first group includes VR engines and CAD engines. Both of them are targeting high-end workstations and VR systems. This was particular true before the expansion of 3D hardware for the PC Market. Even today, VR systems may rely on better technology than mainstream PC. On the other hand, game engines have to run on embedded platforms or middle range PCs. These restrictions oblige game engines to run on specific technology preventing them to benefit from technologies not well spread. If before 2000, academic and institutions may afford US\$ 1 million computers, the fast evolution of mainstream graphics cards have reduce the advantages of developing on supercomputers. Moreover, with the introduction of three new gaming platforms in 2005-6, which offer unprecedented processing power, the main difference between those two groups tends to fade quickly.

Table 3.1: Major modules of 3D real-time simulations.

Component	Usage
Renderer	Display 3D geometry, culling, collision detection
Animation	Skinning, blending
Sound	3D sound, occlusion, reverberation
Input	Game Controller, VR device, touch screen
Physics	Ragdoll, solid, fluid
AI	Behavior, collision avoidance, steering methods
Network	Persistent world, prediction
Kernel	Core functionalities, I/O operations
GUI	Hud, feedback information
Database	Content storage
Scripting	Flexibility, Mod

There is still important difference between the two groups. The game engines are clearly dictated by commercial considerations that force them to offer the best visual and sound experience over their competitors. The money put in these developments is in the ten of millions US\$ [de Gentile-Williams2005]. To compete, VR systems focus their attention on a niche market and high-end systems. However, VR systems even more than game engines need to adopt strict engineering practices to remain viable. When VR systems platforms were costing more than one million dollars, they were hardly any competition. With the drop of hardware costs, the associated software became accessible to a wider audience reducing the overall costs. For instance the license cost for the program Maya went from approximately USD\$ 15000 to less than 5000. Similar cost reductions have affected other tools. The fact the VR systems were able to rely on the overpriced systems, may have caused VR systems developments to give the technological initiative to game engines.

3.1.3 System Architectures Categories

3DRTS architectures come with a set of dedicated requirements. Depending on the needs, the system design will emphasis on providing good potential for easy prototyping of simulations while other systems will try to optimize around technologies. This leads to situations where conceptual approaches may diverge. For instance, military and science applications are generally composed of modules containing advanced algorithms, while games applications may focus on simulations arts. In addition, the scale of the simulations and the targeted platforms will have a great impact on the architecture. Developing a system capable of handling persistent worlds working over the network require particular attention for producing a distributed system including large databases. On the other hand, a system running on embedded platforms may focus on other aspects of the simulation. This great variety in conceptions and requirements are the consequence of great variety in simulations genders.

3.1.4 3DRTS Systems Evolution

The architecture for interactive simulations has growth considerably over the year. The Figure 3.2 depict that early 2D interactive simulations were composed of only few components including 2D renderer and mono sound systems and file I/O operations.

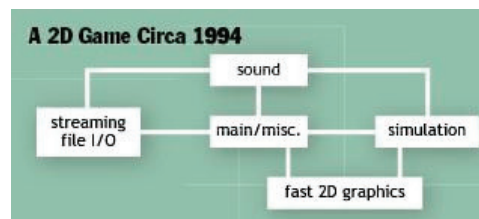


Figure 3.2: Typical architecture used in most 2D games in 1994 [Blow2004].

With the emergence of 3D graphic chipsets for 3DRTS, systems complexity was increased with the introduction of 3D environments, as shown in Figure 3.3. Such applications were forced to handle collision detection within 3D worlds with the obligations to replace 2D sprites by 3D geometrical models.

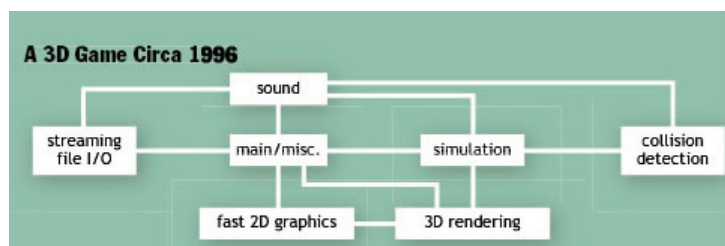


Figure 3.3: Early 3D real-time applications were still mixing 2D with 3D elements [Blow2004].

Over the years, the technological advances with computer hardware have improved the realism and interactivity of 3D virtual worlds by creating environments that are more believable. Virtual characters became capable of reacting to stimuli creating the first steps for intelligent agents. Others aspects of the simulations have also benefits from better technology such as the ability to compute physics reactions in real-time. This increase of complexity is some many elements also force 3DRTS architectures to provide more abstraction layers as describe in Figure 3.4.

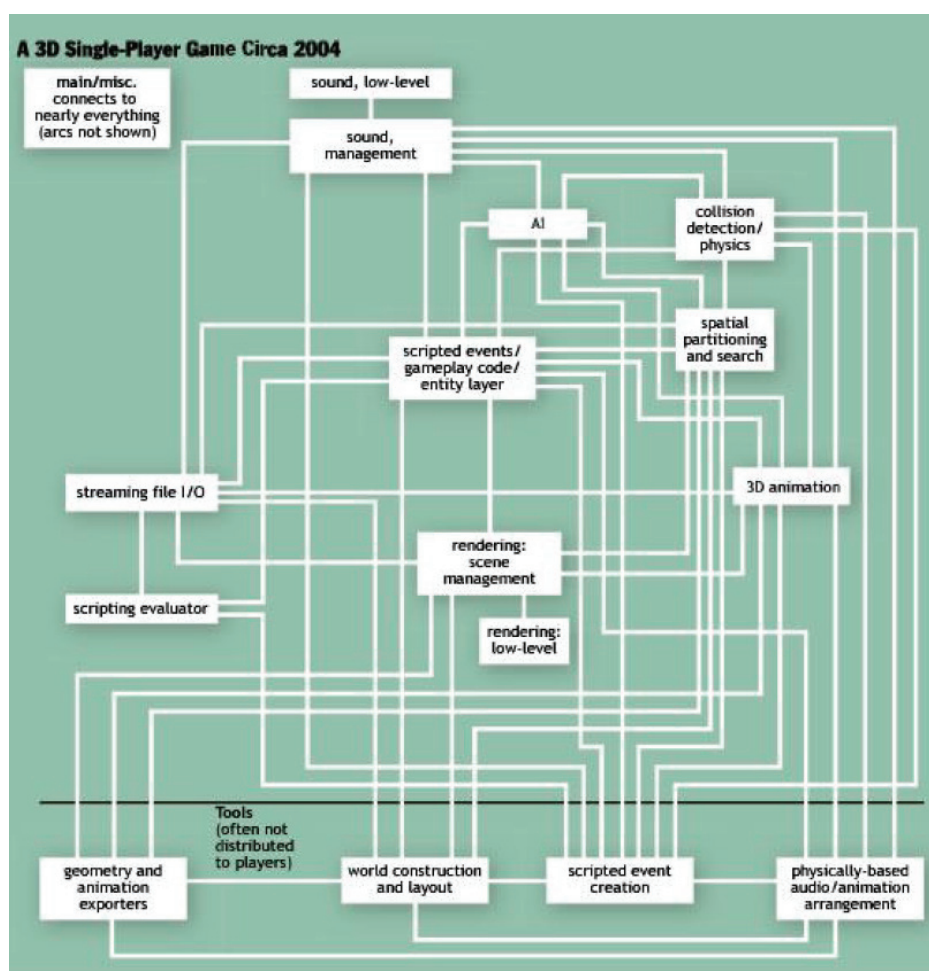


Figure 3.4: Modern 3DRTS systems contain many components that need to communicate in an efficient manner [Blow2004].

With the recent propagation of broadband connections into millions of homes in the world, many 3DRTS are directly connected to the network. Some of the most complex software applications are the MMO, which features persistent worlds with thousand of players in the same environment. The Figure 3.5 depicts the extra layer of complexity involved when developing such systems. They require handling specific modules for servers and clients with a shared middle layer that interconnect them. Only strong developments houses can run and exploit such systems due to their extreme complexity in development, deployment and maintenance aspects [IGDA-MMO2004].

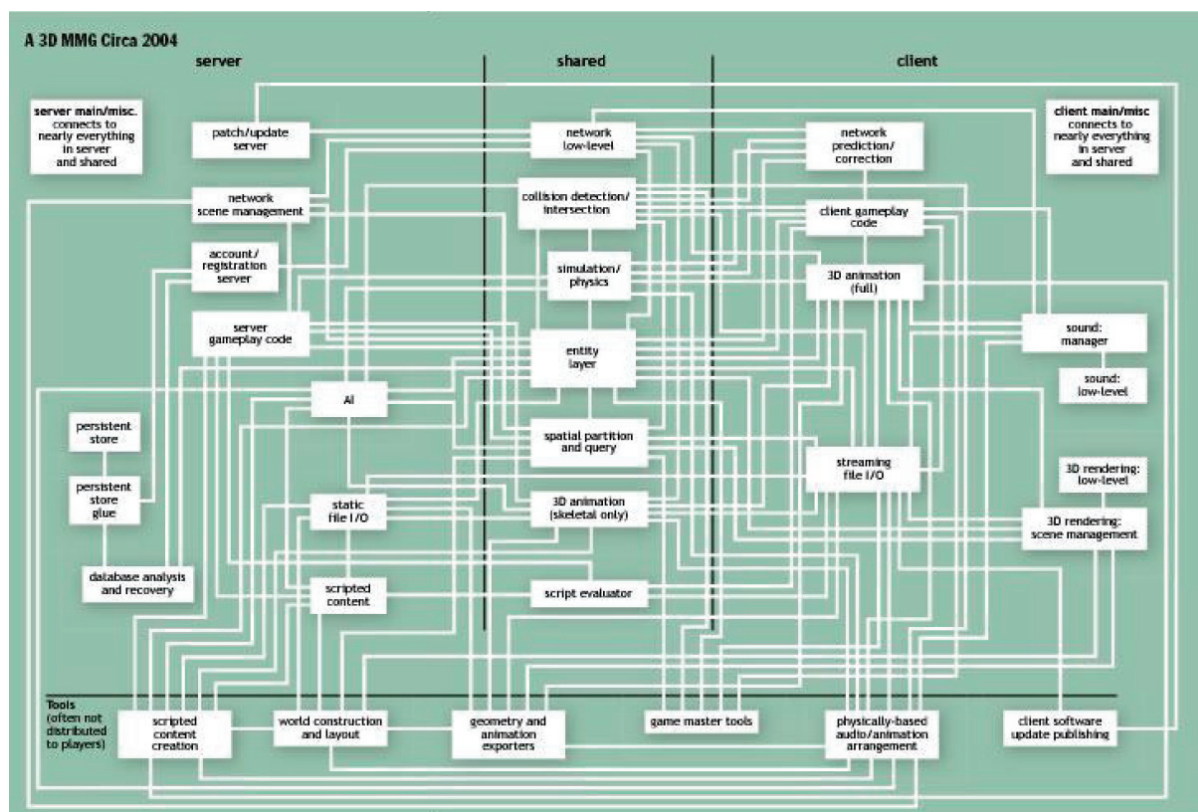


Figure 3.5: A typical MMO architecture design [Blow2004].

The evolution from the first iteration of interactive simulations featuring limited components to the more modern systems is important. Nowadays, small teams using improvised methods cannot anymore develop quality interactive simulations. The needs to control the continuous increase of complexity are dictated not only by pure commercial considerations but also simply by the scale problematic. The future of systems architectures will continue to extend this difficulty with the introduction of heavy parallelism hardware and high-definitions simulations.

3.2 VR Systems

Systems architectures for VR simulations have been developed for handling the communication layers between the software and hardware components. The idea is to promote fast prototyping of VR simulations by providing frameworks and toolkits that separate the simulations to the core system. Many of the existing solutions rely on adopting a single abstraction layer that hides I/O operations for simulations designers. In the next sections, an overview of some existing systems will be given.

3.2.1 Distributed VR Systems

Historically, systems capable of handling interactive simulations were relying on a battery of distributed processors. In such situations, the rendering is purely done in software and is dispatch among the processors. Managing all the mechanisms that will distribute the work is not trivial. The challenges face by pioneers VR developers enforces the need of creating specific systems architectures for distributed VR systems.

3.2.1.1 DIVE

The Distributed Interactive Virtual Environment (DIVE) is a fully distributed heterogeneous VR system where users navigate in 3D where participants navigate in 3D space and interact with other users and applications. DIVE is a loosely coupled heterogeneous distributed system based on UNIX and internet networking protocols within local and wide-area networks. DIVE supports the development of virtual

environments, user interfaces, and applications based on shared 3D synthetic environments. DIVE is especially tuned to multi-user applications. Consistency and concurrency control of common data is achieved by active replication, reliable multicast protocols, and distributed locking methods. Tcl scripts are used to describe ISOs' behaviors. Scripts are triggered by system events, such as timers, signals, or collisions. DIVE applications and activities include virtual battlefields, spatial interaction models, virtual agents, and multi-modal interaction. DIVE is currently available on most UNIX platforms and Windows PCs.

3.2.1.2 dVS

Some VR pioneers including Ford, Gulfstream, McDonnell Douglas, and the U.S. Army Research Lab, were requiring a platform that can scale-up and work in a highly distributed client/server environment. Thus, Division offers dVS, a Unix-based OS layer that supports VR systems on parallel and distributed hardware. Division decided to explore the benefit of software-only solutions that could run on graphic workstations. The goal was to subdivide tasks supporting virtual world's management, into a more fine-grained level than previous solutions, especially for distributed software. dVS allows running each actor on different processors. Separate actors are responsible for image generation (that is, geometry and rendering), spatializing stereo audio, collision detection, Newtonian physics (that is, implementing the laws of gravity, friction, and elasticity on moving objects), and input and output devices control. The latter actor supports various VR headsets and 3D pointing and tracking devices. Another pioneer aspect of dVS is the multi-user aspect of the system. It enables several networked users to work together in the same virtual world. A separate actor, called the Body Actor, provides an interface to each participant in such a shared virtual environment. The Body Actor, assisted by Input and Collision Actors, monitors each participant's position and boundary attributes in real time. This multi-server architecture means that you can run user applications in parallel with the same virtual environment, like for example, to feed continuous data streams into dVS while visualizing them in real time.

3.2.1.3 MORGAN

[MORGAN] is a distributed multi-user framework for VR/AR applications. The goals of the MORGAN framework are to reduce the time-consuming tasks that affect VR simulations developments. The MORGAN framework also intends to provide mechanisms to analyze only few segments of the system functionalities at a time. The framework was developed at Fraunhofer FTT for elaborating distributed multi-user VR/AR applications. Its software architecture design repose on the components-based methodology by adopting CORBA bindings connecting the system with network communications. MORGAN supports many devices such as haptic devices, object tracking, and speech recognition. One strong advantage with this framework is the ability to develop multi-modal user interface that can feature distributed render engine with automatic scene graph synchronizations capabilities. Moreover, the usage of CORBA bindings allows controlling simulations remotely.

3.2.2 Toolkit

The next systems are development toolkits that provide a set of APIs and tools for rapid prototyping of VR simulations. They emphasis on the practical side and do not necessarily focus on presenting high-level architectures designs.

3.2.2.1 SVE

The Simple Virtual Environment [Kessler2000] is a development toolkit that provides an API for rapid prototyping of VR simulations. SVE clearly separate the definition of the hierarchical structure of objects in the virtual worlds from the code, allowing the developers to concentrate only on the simulations rather than on low-level problematic. The emphasis is made on rapid and easy implementation of virtual worlds. The architecture design relies on a layer of abstraction between simulation specific code and devices. SVE provide additional independent extensions for connecting runtime environment such as intensive usage of callback functions. The SVE project started in 1996 and is the results of several iterations of VR toolkits. So far, SVE supported many successful VR applications.

3.2.2.2 World Tool Kit

World Tool Kit [Sense8] is a function library and productivity tools for creating virtual reality simulations. The toolkits come with an API exposing more than a thousand C functions. The WTK provides drivers for a wide range of VR devices and integrates with other products from Sense8 such as a visual builder (WorldUp) and client-server network architecture (World2World) for collaborative capabilities. It provides interactive capabilities through elements such as Paths, Motion Links, and Sensors. Like Performer, WTK is scene graph based. Latest iteration of the toolkit includes support for network-based distributed simulations and CAVE immersive display setups and largest array of interface devices, such as head mounted displays, trackers, and navigation controllers. WTK is written in C and do not rely on OOP, even so C++ wrappers are provided.

3.2.3 Object-Oriented Frameworks

Object-Oriented framework clearly adopt OOP paradigm for the 3DRTS domain. They present a certain level of modularity by being able to represent distinct functionalities into separate objects. Object-Oriented frameworks constitute the mainstream approach with current VR systems architectures.

3.2.3.1 ALICE

ALICE is an object-oriented framework that implements remote collaboration by combining sophisticated computer graphics algorithms, advanced user interfaces and shared session management. This framework was one of the first initiatives to offer flexible prototyping environments for VR simulations. This system architecture relies on coupling C++ object with Python scripts. ALICE runtime engine decoupled the frequency of components so both rendering and simulation loops run on their own flow. The system schedules the different update on a per-frame basis, which provide smoother refresh rate. ALICE development environment features GUIs for runtime management and scripts execution, as well as providing simulation statistics for diagnostics purpose.

3.2.3.2 MAVERICK

MAVERIK [Advanced_Interfaces_Group1999] is a object-oriented framework targeting VR simulations of large-scale virtual environments. It is part of the DEVA framework that extends powerful interaction and visualization capabilities of MAVERIK with higher level operating environment supporting multiple users in context of distributed shared virtual environment simulations. MAVERIK is program in C but simulate the object-oriented encapsulating and polymorphism patterns. The system is designed to be open-ended in the way that it represents different kinds of models. It uses callbacks for importing and converting data structures without forcing client users to use predefined data structure representation. Also to the contrary of most VR frameworks, MAVERIK rely on immediate mode rendering as opposed to the widespread retained mode rendering approach provided by scene graphs. MAVERIK architecture reposes on the micro-kernel design pattern and do not employ scene graph representation that do not always perform well for large-scale models. The framework uses a single representation combining both graphical and application data. This idea is based on grouping methods operating on the same datasets (structures holding pointers to sets of C callback methods for display, calculation of bounding volumes, selection/manipulation). However, the clear design still help for reusing large-scale design by decoupling modules into separate callback functions. Developers are provided with runtime customizations capabilities that affect directly dynamic exchange of particular mechanism such as culling, collision detection, or rendering. The C nature of the framework reflects the difficulty to adapt the MAVERIK for rapid prototype of VR applications.

3.2.3.3 JADE

JADE (Java Adaptive Development Environment) [Jade] is a component framework of Networked Virtual Environment Systems [Oliveira2002]. Based on the Java platform, JADE is influenced by the concepts developed by BAMBOO [UCSC]. Taking advantages of mechanisms like reflection and late bindings available natively in Java. JADE targets system abstraction layers. JADE focuses essentially on software development and do not address issues related to the authoring or content creation pipeline. Jade rely on the micro-kernel architectural pattern. Applications are expressed in form of hierarchy of modules components, similar to concepts proposed by [Arnaud1999, Dachselt2002] and NPSNET-V component framework [Capps2000]. JADE introduces context-based and container-based composition model, which is uncommon for 3DRTS systems,

where composition is predominantly connection or data-driven (see NPSNET-V). Every JADE component is derived from a module abstract class used for enabling late binding mechanism. Modules are controlled by the JADE micro-kernel using a modules manager. Active modules can be powered by the micro-kernel or use their own internal threads. This allows spreading the workflow but still guarantee that the proliferation of threads remains minimal. Each module has to implement a plug-in interface providing a set of functions such as `initialise()`, `shutdown()`, `activate()`, `deactivate()` and `buildDescription()`. The modules managers perform the runtime lifecycle management based on this plug-in interface providing mechanisms for dynamic loading or modules at run-time. Preventing, the classical bottleneck module communications that are used for traditional caller-callee collaboration, the JADE framework used its reflection mechanism interface for discovering them at runtime, that allow for late binding of collaborations. JADE also introduce an event model based on the publisher-subscriber design pattern. Events carry information such as ID, source and inner type. JADE provide support for synchronous and asynchronous event processing. Finally, JADE relies on XML configuration files for configuring the system but lack the support of an interpreted scripting language. So far, the applications built on top of the JADE framework were confined to VRML and Java3D environments. However, many concepts presented are applied by other systems architectures.

3.2.3.4 DIVERSE

DIVERSE (Device Independent Virtual Environments: Reconfigurable, Scalable, Extensible) [Kelso2001] is an open-source object-oriented framework augmenting functionality of SGI Performer to facilitate developments of input/output device independent VR systems. It allows for dynamic selection and configuration of various navigations and interactions techniques. DIVERSE separates applications from the input/output hardware specifics, such as CAVE and HMD devices. The goals of the DIVERSE framework are to allow building applications that will run on desktop computers as well as various immersive systems. DIVERSE framework ideology is to prevent the “center of the universe” paradigm by allowing the framework to interact with many other APIs and toolkits including Performer, OpenSceneGraph, or OpenGL as depict in Figure 3.6. DIVERSE define set of independent mechanisms use to extend the system. They are based on a concept of augmented abstract classes that define plug-in interfaces for dynamic loading of extensions. Plug-in interfaces serve as configuration modes that are powered by the DIVERSE runtime-engine. This allow to loaded and unloaded modules using late binding mechanisms. Once loaded the extension are connected into a scene graph, which reproduce the behaviors of the Performer execution models base on pre, post cull, and draw steps. However, one limit of the DIVERSE framework is that it does not provide abstractions for bottleneck interactions or behavioral coupling functionalities.

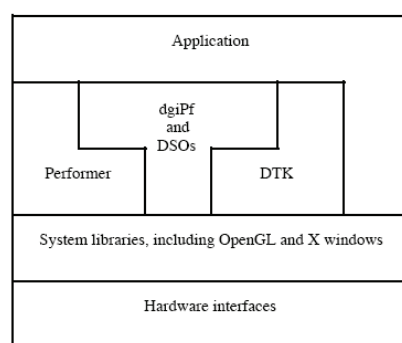


Figure 3.6: DIVERSE libraries encapsulation.

3.2.3.5 LIGTHNING

LIGTHNING [Dowd1996] is highly flexible object-oriented applications framework that offer quick developments of VR simulations by coupling C++ objects into TCL scripting layers. The Figure 3.7 describes the system architecture organization.

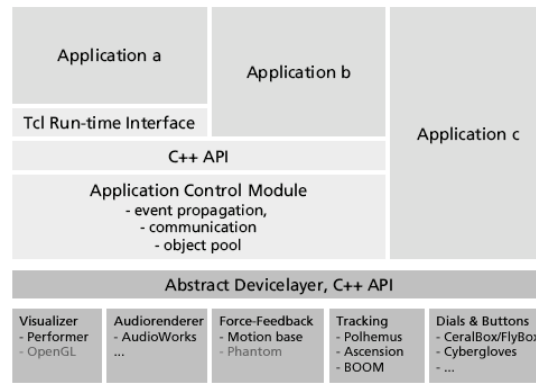


Figure 3.7: The LIGHTNING system architecture.

The LIGHTNING framework enforces the communication between modules by propagating events directly to the ISOs. The collaboration between modules is following a similar events graph as seen in the VRML approach. Every object is directly coupled to system functions as described in Figure 3.8.

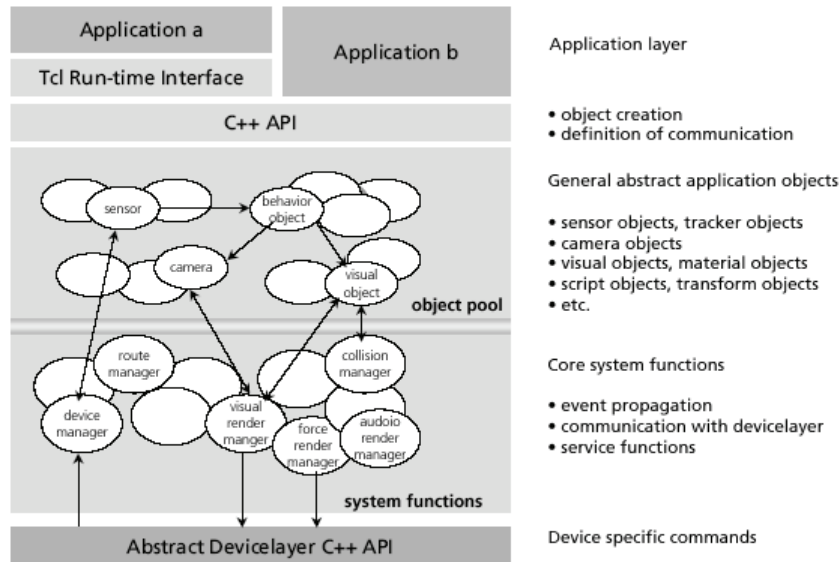


Figure 3.8: Objects are directly coupled with system functions.

LIGHTNING aimed at a broad range of applications domains and it is not restraint to a single language. It provides mechanisms for decoupling rendering and simulation updates. Moreover, it allows subbing tasks to different processors for improving the overall system workflow.

3.2.3.6 VIRPI

VIRPI [Germans2001] is a high-level toolkit for interactive scientific visualization in virtual reality. This toolkit provides communications, measurements paradigms, and interactive VR. The original idea of the VIRPI framework was to excerpt that users want to interact seamlessly with a simulation, understand its mechanisms with limited data manipulations. The VIRPI toolkit address this problematic by coupling the framework to CAVestudy (see Figure 3.9). CAVestudy wraps an unaltered remotely running stateless simulation program, and presents a simple programming interface to the user in the VR environment. This way, the user can connect buttons and sliders in the virtual laboratory to parameterize the simulation, and read out the data from the simulation. The VIRPI framework hides the detail implementation into a low-level layer called Aura. Aura presents a lightweight interface to underlying low-level libraries. Without noticeable decrease of efficiency, it adds features that the underlying software lacks, or it simply passes calls in a transparent way. For managing the scene, Aura uses a scene graph representation similar to the IRIS Performer implementations. By

having the opportunity to combine either CAVElibrary or VR Juggler, providing simple interfaces for input device and rendering contexts, VIRPI and Aura present coherent interfaces to different hardware. However, the framework does not handle issues like multi-pipe output, multiprocessor systems and shared memory.

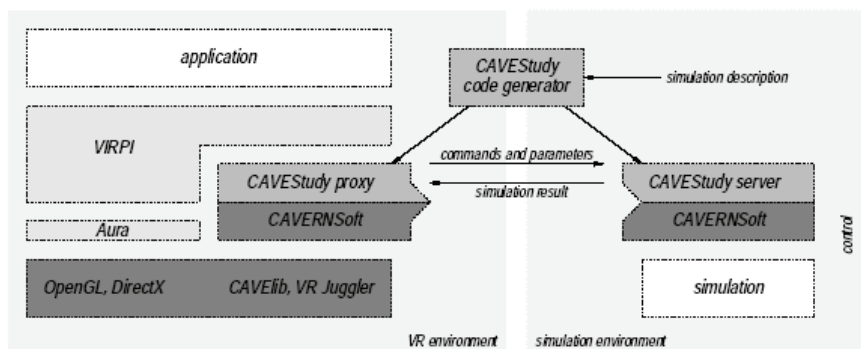


Figure 3.9: The VIRPI, Aura, and CAVESStudy interconnections.

3.2.3.7 VR Juggler

Dr. Carolina Cruz-Neira and a team of students at Iowa State University's Virtual Reality Applications Center started the VR Juggler project in 1997. In November 2003, Infiscape Corporation began offering commercial support and services for users of VR Juggler. Since then, Infiscape Corporation has assumed responsibility for maintaining, improving, and stabilizing VR Juggler to ensure its usability in production environments. Faculty and students at the Iowa State University Virtual Reality Applications Center perform research-oriented and long-term visionary work to keep VR Juggler on the cutting edge of virtual reality technology. VR Juggler architecture design relies on the object-oriented programming methodology. The framework supports the development of VR systems featuring various configurations of low-end and high-end input/output devices. VR Juggler is scalable for a wide range of hardware from desktop systems to complex multi-screen systems running on clusters or on high-end workstations and supercomputers. The portability of the VR Juggler code ensures that any OS can be used. VR Juggler supports IRIX, Linux, Windows, FreeBSD, Solaris, and Mac OS X. VR Juggler aim at provision of flexible middleware layers that clearly separate application dependant code from hardware specifics implementations. It targets the separation into tier that underlines this separation as describe in Figure 3.10. VR Juggler support rapid prototyping of applications with the help of dedicated tools and GUIs that interact with many VR devices that can be added or removed at runtime. Some key features of VR Juggler include its attention to address performance, scalability, and faults tolerance with efficiency. The architecture design focuses on implementing the micro-kernel design pattern. The role of the kernel is to abstract all OS systems level mechanisms including the ones use for concurrent systems such as multithreading, mutex and semaphores. Components are managed by its micro-kernel, which provide late binding mechanisms allowing adding and removed components dynamically. VR Juggler also emphasis on adopting design pattern with special usage of the manager pattern for handling environment and rendering display. They manager are used as bridge for object communications. Managers in VR Juggler are declared as plug-ins interfaces that allow them to collaborate with each other. However, VR Juggler do not prevent messaging bottleneck affecting components. Since Managers do not know about each other, handling of dependencies is simplified and the late binding mechanism can plug and unplug them at runtime. Another fundamental aspect of the VR Juggler framework is the methodology of manipulating simulations objects specifically through tasks schedulers. This implies that the kernel can separate the simulation objects from the underlying OS but also they cannot access directly managers. The kernel keeps full control on the simulation objects. This indirection allows for runtime reconfiguration, replacement, adding, removing of internal managers. On the other hand, Application Objects can access directly External Managers like Draw or Sound Managers that provide access to the specific graphical or sound APIs. In effect, VR Juggler separates Applications Objects from OS and input/output hardware specific but not from the APIs of various toolkits supporting rendering, sound, etc. VR Juggler provide advance GUIs for controlling the system and come with a XJL (extensible Juggler Language) configuration mechanisms acting as an XML declarative scripting language. However, VR Juggler does not provide behavioral coupling scripting languages.

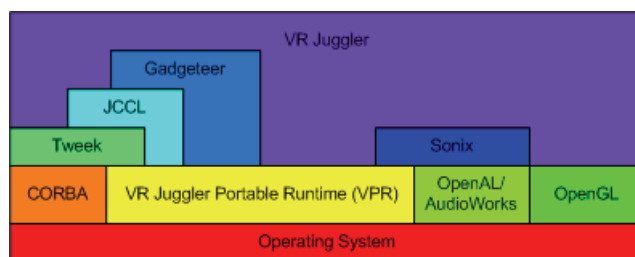


Figure 3.10: VR Juggler multi-layer system architecture.

3.2.3.8 Virtools

[Virtools] is a development environment that bundles together models and code. Virtools allow to prototype 3DRTS very quickly and effectively with limited programming knowledge. The technology includes authoring tools, a behaviors engine, an embedded scripting language, a rendering engine, a web player and a SDK. The Figure 3.11 depicts the components workflow. Virtools promote authoring by offering the ability to combine assets (graphics, sound, animations...) together by simply clicking and dragging shapes inside an editor. However, assets cannot be created directly and this requires the use of external DCC tools. A strong element of Virtools is its behaviors engine that controls all the interactivity and simulation logic. Virtools clearly separate objects, data, and behaviors by creating assembling objects that interconnect the logic and data layers. In Virtools, a behavior is a collection of attributes and their values are connected to ISOs, so they behave a certain way. Therefore, it becomes possible to build many behaviors with not programming interactions allowing to quickly prototype VR and games applications. A specific action manager enhances the authoring by letting developers creating scripts for frequently used functions.



Figure 3.11: The Virtools components workflow.

Virtools presents a multi-layers architecture, where the end-users are interacting only with the top layers, which is broken into self-contained functions. This reduces the knowledge required to understand the underlined mechanisms. In addition, this gives the ability to equip the platform with dropping and dragging capabilities, where each high-level function is directly represented with corresponding buttons and labels. Virtools also contains a rendering engine, which emphasis on providing high-level mechanisms to control ISOs in the 3D scenes. It support Vertex and Pixel shaders' effects and can be replaced by other rendering engines if

needed. One particularity of Virtools is to provide a web player for playing simulations. This web player is very effective for creating tutorials as modifications can be seen in real-time immediately. Moreover, the web player is freely available, which help to disseminate 3DRTS made with Virtools. To create new behaviors or modify existing behaviors, Virtools include a SDK that provide access to the top most level of both the behaviors and rendering engine. This gives the opportunity to write specific plug-ins and extension in C++. In addition, Virtools is equipped with a custom scripting language inspired from the C programming language. The strong aspect of Virtools rely on his abilities to prototype 3DRTS with little interaction from core programmers. If Virtools is principally used as a prototyping developments framework, many successful commercial projects were also created (Siberia, Jack the Ripper...) [Virtools]. Nevertheless, Virtools may lack important features in some areas, in particular for handling characters animations. For instance, it is not possible to play multiple animations for each frame of animations, forcing to employ only one basic animation at a given point. The underlined design prevents low-levels modifications to ensure a smooth learning curve and to ease prototyping 3DRTS for non-programmers. Therefore, Virtools may not always provide enough flexibility for applying researches and experimentations on low-level mechanisms. Currently, Virtools runs exclusively on Windows powered platforms (PCs and Xbox) and is integrated seamlessly with assets management's applications such as Alienbrain [Avid-Alienbrain].

3.2.4 Component-Based Frameworks

Component-based frameworks are slowly increasing in popularity for describing the modules and elements forming 3DRTS. So far, only a restricted number of frameworks adopting this methodology have reached the deployment phase of development.

3.2.4.1 I4D

I4D is a component framework [Geiger2000] focuses specifically on VR/AR system developments including both content and software side. I4D provide visual authoring environments for developing the system (see Figure 3.12). I4D components are called actors that encapsulate both visual (cameras, lights, 3D objects, virtual characters) and behavioral (e.g. keyframe replay, sound replay, look-at, follow-object, physic, ARToolKit visual camera tracking). Each actor features set of attributes such as matrix, LOD, etc. The scene management is similar to standard scene graph approaches, and is used to represent both virtual environments and application compositions. From this perspective, I4D shared many similitude with concepts promulgated by [Arnaud1999, Capps2000, Dachsel2002, Oliveira2002]. I4D actors provide data-driven collaboration mechanisms. Each actor can publish string-based messages that can be of single cast, multicast, or broadcast natures. I4D forbid explicit connection between actors, and in effect, I4D feature data-driven programming and composition model, where collaborations between components are mediated through string-based messages. The structural coupling of components is realized using authoring tools for creating scene composed of actors. Their attributes are accessible within the authoring tool, which export a XML declarative script. ISOs coupling can also be provided dynamically through Tck/TL scripts. The I4D execution environment allows for dynamic loading and unloading of components, priority based scheduling of action execution. I4D also separate the system from the underlined OS and 3D graphics libraries through an I/O layer. Typical applications features path finding for robots, or evolution of 3D creatures using bone articulations. However, the high granularity of the system as well as endorsing a full text based messaging approach may be a limitation factor for more intensive 3DRTS.

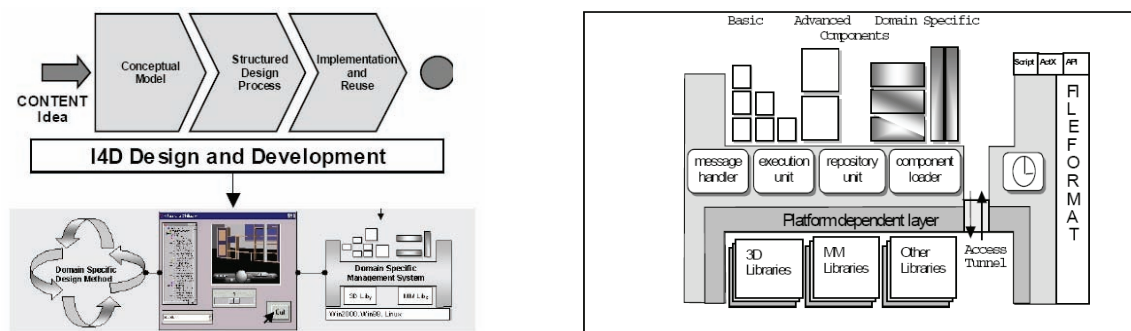


Figure 3.12: The left picture represent the I4D design and development flowchart. The right picture describes the component layers featured by the I4D system architecture.

3.3 Game Systems

The development of 3D games is one of the most complex software to develop. They have to optimize the current technology and continuously improve their system to match the high level of competition and users expectations. Most game systems developers do not explain their conceptual approaches. Nevertheless, the next section tend to draw an overview of some technical decisions taken by game developers based on freely available information and direct communications with game systems engineers.

3.3.1 VR Systems vs. Games Systems

VR Systems and Games Systems are very similar. Both are relying on specific hardware and have to optimize the resource usage for both fast rendering and interaction of complex virtual worlds. The principal difference is that game systems are created for mainstream personal computers and home console platforms, where VR systems may benefit from high-end systems [Lewis2002]. Moreover, VR systems are generally connected to VR devices such as HMD or haptic stations. These differences, which were once important when the performance of workstations for 3D graphics were considerably outperforming mainstream computers is now gone. Few years ago, the difference was several orders of magnitude in the capabilities of rendering more polygons, applying richer textures and fill rates limitations. Nowadays, VR systems hardware remains more advanced than the average home computers, but the difference is more marginal. In effect, VR systems may not anymore rely on pure processing throughput advantages but have to promote new interaction models using VR hardware, which are still very expensive and fragile. The rapid developments of graphics processors for mainstream computers give us the ability to directly compare both VR systems and game systems in the same hardware category. If difference in technology do not defined anymore theses two categories, we need to consider the difference from conceptual perspectives. Game systems intend to provide the best visual effects with action-oriented interaction with their environments. Their goals are to entertain a wide audience of users by pushing hardware limitations. As the competition is tight, optimizations and quality are crucial for the project viability. On the other hand, VR systems have different goals. They are focused on high-end systems with a much smaller user base. Since the customer's needs are specific, they may overcharge their software solutions making VR systems more livable business. Thus, VR systems have to provide functionalities to prototype a much wider range of simulations base on the same core technology. Another distinction is that VR systems and researchers focus on conceptual ideas and do not suffer from the same set of constraints than game developers. However, they all do intend to create living virtual environments that reflect some aspect of the reality. We can observe that the two fields as becoming closer to each other as some research algorithms for 3DRTS are developed on similar hardware making them livable for commercial integrations [Bungie2005 863]. Some initiatives like the Serious Games [Serious-Games] promote games for edutainment purpose while some organizations such as the US Army commissioned the development of PC games promoting their operations [Casey2005]. Such from a software architecture design, theses two fields are working toward the same ideal, which is to provide tools and methodologies that will unleash the potential of current software and hardware.

3.3.2 3D Graphic Rendering Centric Engines

During the expansion of 3D graphic hardware, the fast evolution and complexity to create efficient 3D renderer had provided the ability to few software developers to capitalize on the idea that a graphic engine could serve as a core component for 3DRTS. One of the most notorious game developers to license his technology was id Software with the Quake and Doom engine [idSoftware]. id Software was using his own games as a showcase of the capabilities of their technologies, which started as a 3D renderer only solution and later add network capabilities and objects management. As the company was leading in 3D renderer engine, they managed to license their game engine up to 250,000\$. However, the risks and costs involved to develop an equivalent 3D renderer were much higher. Nevertheless, the solutions provided by idSoftware do not represent an exception. For instance, open-source initiatives such as Ogre (Object Oriented Game Engine) [Streeting], Irrlicht [Gebhardt], Crystal Space [Crystal_Space] or Ca3D Engine [Fuchs] concentrate their effort for providing a fast 3D renderer. They are intended to be connected to different modules managing the others simulation aspects like physics or AI management. The advantages of these stand-alone 3D renderers are that they offer a better flexibility when building systems architectures by being able to carefully choose the appropriate components.

3.3.2.1 Ogre3D

Ogre3D [Streeting] is an object oriented 3D graphics engine. It uses OO interfaces designed to minimize the efforts required to render 3D scenes, by abstracting 3D implementations such as OpenGL or DirectX. The system relies on the ability to plug specific extensions that handle particular element of the 3D graphic rendering pipeline. Such plug-in include scene managements, where different implementation can be provide from simple view frustum culling to more complex BSP or Octree. Others plug-ins are also available for extending the engine with additional node kits such as particle effects or shaders. The Ogre3D architecture design is very clean and well documented. Steve Streeting, the main contributor of the engine rationalizes the design decision by specifying that the purpose of Ogre3D graphic engine is not to allow developing simulations faster, but more to build a more flexible graphics component that can be reused for many different situations. In addition, integration with other components is facilitated. Currently, the system runs on major platforms including Linux, Windows, or even home consoles. A strong characteristic of Ogre3D is its material declaration language. This language allow to maintain material and shaders assets outside of the code which allow to use shared effect files on geometry similar to the .FX files from DirectX [St-Laurent2005]. Ogre3D graphic engine capabilities include support for recent technologies such as dynamic shadow, normal maps, etc. It also provide many tools that help to export geometry from 3D modeling packages as well as the ability to generate on the fly progressive meshes, which was extend in [de Heras-VSMM2005] for creating LOD for our virtual characters. The engine also provides flexible mesh data formats, which are combined to skeletal animation, including blending of multiple animations compute on the GPU. Ogre3D also rely on metadata file using XML tags, which are independent of the final binary files format used at run-time. This help to create exporters that are independent of the engine. As Ogre3D is freely available under the LGPL license, developers can extend the engine functionalities. Many institutions and academic courses are using Ogre3D for lectures on 3D programming due to its clean and extendible design. However, Ogre3D is not a simulation framework, as it focuses only on delivering a 3D graphics engines. Thus, Ogre3D can be used as a graphical component within a component-based system.

3.3.2.2 Torque Game Engine

The developers of the Torque game engine focus on providing a cheap commercial solution for building 3DRTS [GarageGames-Torque]. The license cost only 100US\$ for each developer. The Torque game engine features good supports from homebrew developers. The system usage is made by exposing functionalities directly within a custom scripted language. Contrary to many projects where scripting language is directly embedded into the system, the approach of the Torque game engine is to control the whole system within the scripting language. The default behavior is to provide a Shell that allows to dynamically loading modules on request. Thus, the simulation editor is directly built into the engine, allowing editing world parameters into a true WYSIWYG GUI editor.

3.3.2.3 Crystal Space

Crystal Space [Crystal_Space] is a portable 3D graphic engine and development kit written in C++. It supports features such a true six degree of freedom, shaders, portals, etc. Crystal space also support skeletal animations based on the [Cal3d] animation library, as well as partial scripting support using either Python, Perl Or Java. It also incorporate physics plug-ins based on ODE [Smith]. Crystal Space runs on GNU/Linux, general UNIX, Windows, and MacOS/X, specific platform optimizations are use such as NASM or MMX instructions. Crystal Space is available under the LPGL license. Some key features of the Crystal Space engine include the use of a light-weight component mechanisms called SCF for shared class facility, that allow flexible creation of components in a C++ environment. This approach is generally better suited than standard enterprise-scale components approaches commonly use in other domains. The primary purpose of the SCF is to separate the interface from the concrete implementation of a component or plug-in as they are seen by the Crystal Space architecture design. This clear separation makes it possible to replace component implementation with updated version or using different set of low-level libraries depending of the targeted platform capabilities. The SCF architecture is a well define mechanism relying on interfaces. Each interface represents a collection of abstract methods and serves as contracts specifying the access points and functionalities of a component. Using theses interfaces, the Crystal Space system manager can discover plug-ins automatically and load them dynamically by providing Meta information about each plug-in capability into XML files defining the plug-in specific settings. In addition, the Crystal Space architecture provides the ability to define a global access point to components, which reduce the mutual dependencies between components. However, even so Crystal Space framework

provides a modularization base on components, the framework confine its capacity in the domain of 3D rendering and additional elements such as AI or scripting capacities have to be provided by client developers.

3.3.3 Component-Based Game Framework

3D renderer engines were very successful in the early ages of 3D hardware. Nowadays, the need to provide better and high-level design became more critical. Some game developers decided to build component-based software that would provide more elements and components for better developments processes. One of the first contenders was the resulting work from the developer Epic Games. They first created the Unreal Game Engine, which becomes quickly a popular engine. Similar to their competitor, Epic Games was developing their engine for their own software before considering licensing their technology. Rather than focusing only on 3D rendering, the Unreal Engine technology had evolved into complete system architecture for building 3DRTS. The latest iteration called the Unreal Engine 3.0 feature state-of-the-art 3D renderer for next-generation hardware but also integrate many middleware components and content creation exporters. These tools allow controlling the system from a higher-level perspective. If in the past iteration, the Unreal engine was mainly dedicated for FPS (first person shooter) gender, the current version is very versatile and the current software based on this technology goes beyond this gender.

3.3.3.1 RenderWare

Criterion software with its RenderWare [Criterion] solution is perhaps the actual commercial engine that rely the most on the component-based design. The RenderWare framework relies around a set of modules and toolkits that provide the abstraction layers for creating fully cross-platforms software. The framework provides several components such as a 3D renderer where a concrete component implementation is available for the principal platforms such as PC, Gamecube, Xbox, or PS2. The same approach is used for AI and Physics component (see Figure 3.13). It does this abstraction not only on the programming side but also on the content creation by emphasis on storing the information into metadata. This allows exposing assets for each platform with minimal efforts. The success of the RenderWare platform with the current generation of software was unprecedented with up to 20-25% of the commercial games directly built on top of this technology. This is mainly due to a very well designed and flexible architecture rather than on the capabilities of its internal components. Presented as a computer game middleware platform, RenderWare is design around a component-based architecture, which consists of reuse, and interchange software components without imposing strict architectures and structures to clients' developers. The RenderWare middleware is combine a platform integrating technology suite for graphics, physics, AI, audio, etc. into separate components and a set of tools and editors called RenderWare Studio. The RenderWare Studio consists of providing all the toolset for developing simulations including assets management and cross-platforms content creation pipeline. So far, RenderWare was used in more than 500 commercial published games, including some AAA titles such as Sonic Heroes (Sega), the Movies (Lionhead Studio) or Gran Theft Auto: San Andreas (Rockstar).

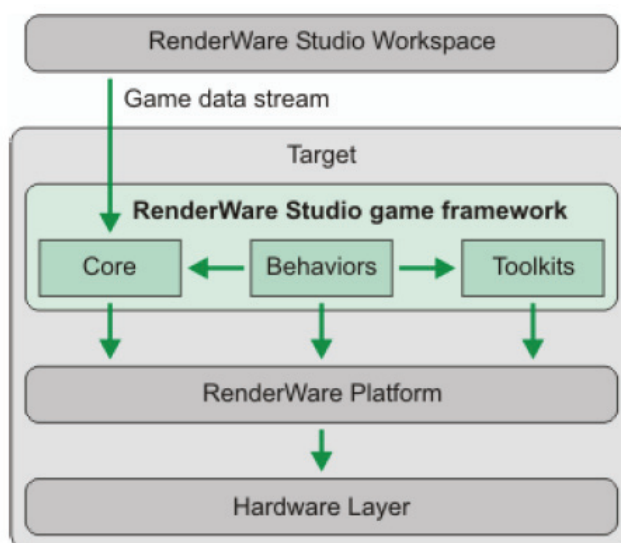


Figure 3.13: RenderWare multi-layer architecture.

More interesting are the tremendous resources capacities of the development team. Criterion Software employs more than 250 people spread into six sites (Austin, Guildford, Vancouver, Derby, Paris, and Tokyo) in its software technology division. One purpose of the RenderWare platform is to provide a generic framework that can be used to develop any simulations. As an outcome, the technology involve in the RenderWare platform may not directly compete on a feature by feature comparison with more specialized engines, but this at the rational cost for high flexibility. In addition, as RenderWare is constantly improved, radical changes are not allowed to ensure stable environments for clients and to be more responsive to the different particularities of each supported platform. The RenderWare platform is supplied by on-the-shelf components:

- *RenderWare Graphic*: a cross-platforms renderer with specialized optimizations.
- *RenderWare Physics*: for fast modeling of physics simulation.
- *RenderWare AI*: multi-platforms AI middleware.
- *RenderWare Audio*: real-time audio for multi-platforms simulations.

The RenderWare platform components are designed to work together, but can be exchanged for other third-party components. The strong advantage of the RenderWare middleware over alternative solutions is that it includes the RenderWare Studio, which includes most of the tools required for handling cross-platforms resources management (see Figure 3.14). This include design tools for creating simulations objects attributes, and controlling simulations events, as well as the abilities to build environments and store the information into optimized run-time libraries. The middleware is also integrated with the asset management toolkit Alienbrain [Avid-Alienbrain].

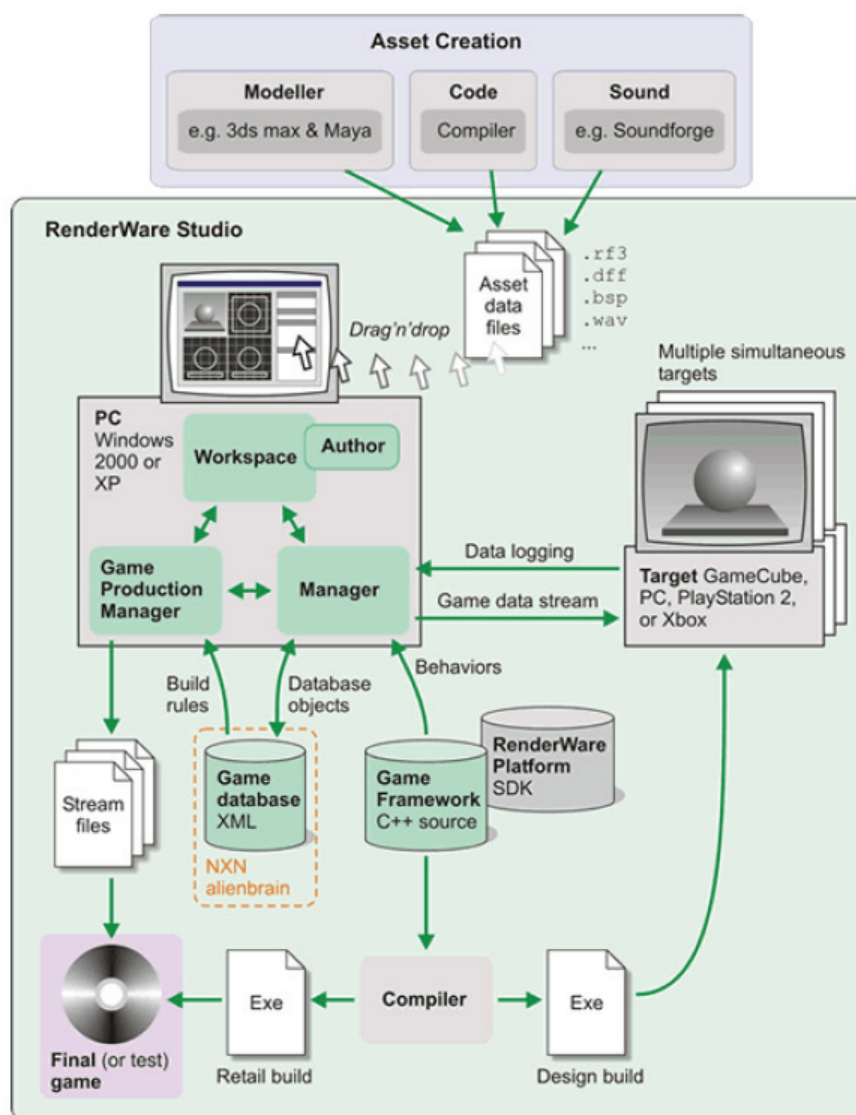


Figure 3.14: RenderWare studio overview.

3.3.3.2 Unreal Technology

Epic-games had develop the Unreal Technology [Epic-Unreal] over the years. The latest installation of their game framework is called the Unreal Engine 3. This framework is complete game development engine targeted for next-generation of home consoles and PCs. The Unreal Engine provides different rendering implementations with a particular focus on DirectX-equipped hardware, providing a vast array of core technologies. The Figure 3.15 depicts the software dependencies of a typical application relying on the Unreal Technology.

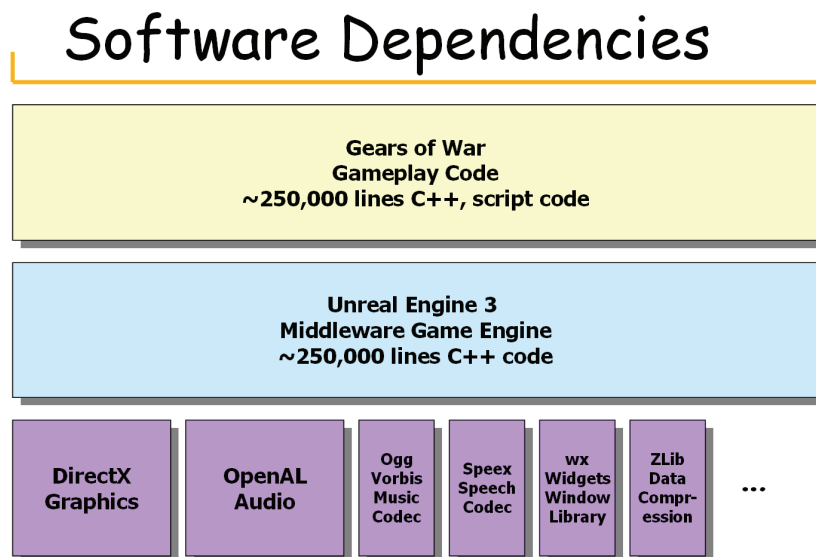


Figure 3.15: Software dependencies use in the video games Gears of War [Sweeney2006].

One design aspect of the Unreal Technology is to ease the content creation by putting as much power as possible in the hands of artists to develop assets using visual environments with minimal programmers' assistances. The architecture relies on high level of modularity allowing creating simulation specific extensions. From a technical perspective, the Unreal Technology support all recent 3D graphics rendering techniques such as high dynamic range lighting or advanced dynamic shadowing. The framework also offers visual interfaces for managing complex materials like displacement maps or normal maps and others shaders, either directly into 3D modeling packages or into using proprietary tools. Using these editors, artists are able to generate terrain, vegetation, or LOD for geometry. The rendering of virtual characters is also a central element of the technology, with the ability to scale the characters for both crowd simulations or for high fidelity actors. The virtual characters animations system support up to four bones influences per vertex, as well as providing data-driven body parts controllers and inverse kinematics. The Unreal Technology relies also on the component-based approach and use middleware software for handling physics [Ageia] or vegetation [IDV]. This flexibility allows offering state-of-the-art framework encapsulating the best software components available. The AI component provides an object-oriented framework, which defines entities (agents, vehicles, etc). The AI system is a multi-level architecture, where decisions can be supported for individual or group of agents, as well as handling short-term and middle term memory. To control the simulation behaviors, the Unreal Engine framework uses a custom interpreted scripting language called UnrealScript, which is connected to the UnrealKismet visual scripting editor that provide the ability to communicate with the system in a data-driven approach. From a more concrete design perspective the Unreal Engine 3, present an object-oriented C++ engine with software framework for persistence, dynamic loading of code (UnrealScript) and content, as well as strong portability, and debugging mechanisms. The engine clearly separates the simulation code from the core engine allowing developing them in higher-level languages featuring garbage-collection and imperative programming style. In effect, ISOs are manipulated through the UnrealScripts providing the support for handling metadata and simulation events. In contract the numeric computation rely on low-level and high-performance code handling scene graph traversal, collision detection... taking full advantages of computer hardware intrinsic functions such as SIMD instructions. The complete code base provides unified interfaces as well as the abilities to spread tasks on several threads, such as suitable multithreading background streaming of resources. The engine required to manually synchronized data and there is not compile-time verification for deadlocks or race conditions. The general threading model consist of a main thread responsible for all the serial code, a heavyweight rendering thread and a pool of 4-6 helper threads dedicated for simpler tasks (see Figure 3.16).

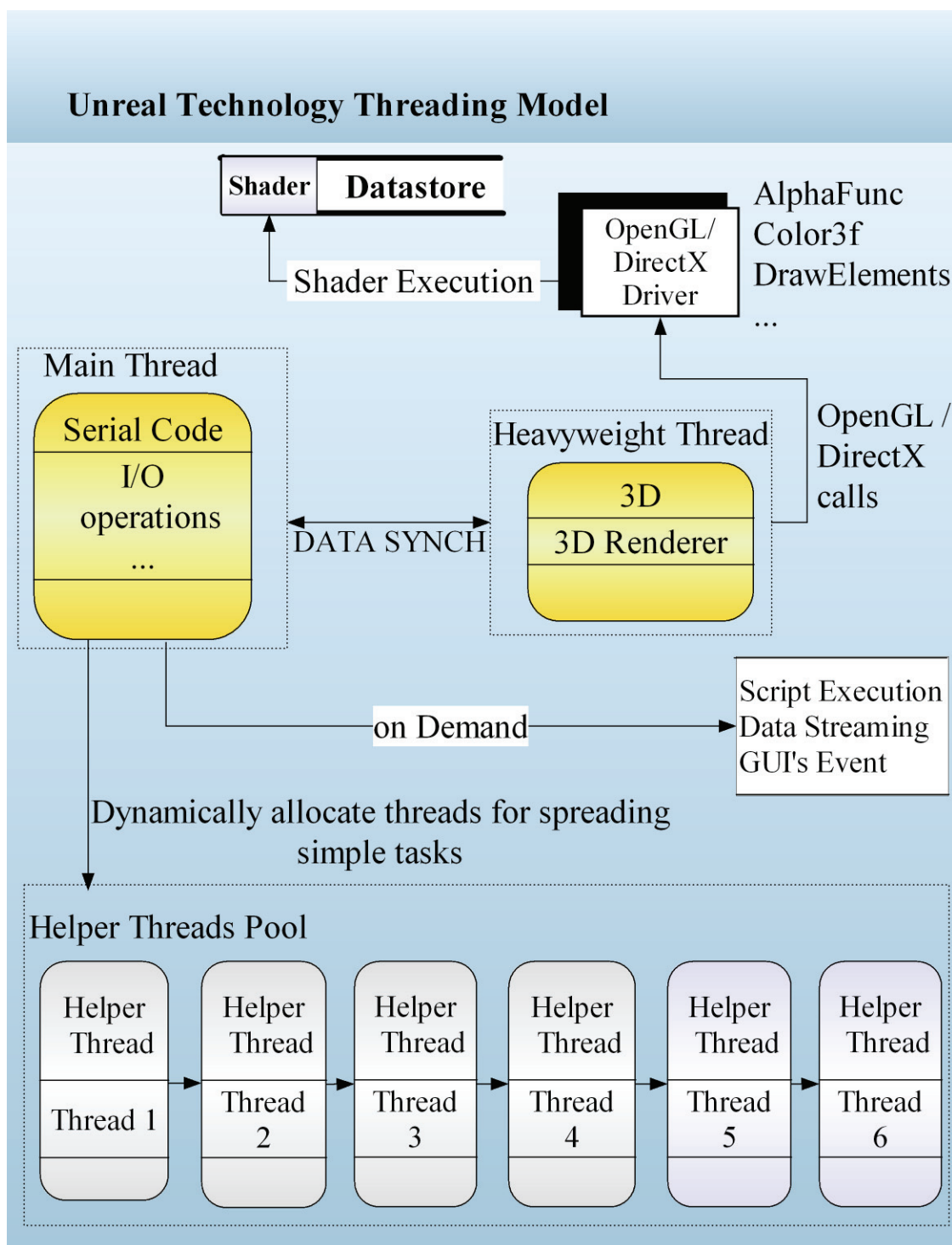


Figure 3.16: Unreal Technology threading model.

3.4 Conclusion

This chapter described some existing frameworks for 3DRTS with their particularities as well as presenting some generic methods. The next chapter will enter into more details concerning design patterns and approaches that affect directly the development of 3DRTS architectures.

Chapter 4

Scope and Structure of Work

This chapter will introduce the different architecture styles considered that fulfill a set of pre-defined requirements, which include good flexibility, extendibility and maintainability traits. The discussion will also illustrate how to improve and handle scalability, variety and reuse aspects for both code and content.

4.1 Design Principles

[Jones2003] has summarized 20 years of software research developments and the relations of proper balancing between design and construction phases based on the type of applications. They showed that 3DRTS are considered as mission-critical systems, which refer to systems that have to optimize the usage of available resources on dedicated hardware architectures. [Jones2003] and al also describe that such systems do require extensive planning as well as providing flexible layers to constantly adapt systems to new requirements and technology. Successful mission-critical systems are built around strong architectures. As of today, many current mission-critical systems continue to rely on building software with on the fly methodology or poor usage of software engineering practices [McConnel1996]. The needs of developing more mature software architecture for interactive simulations require handling the problematic from a higher perspective. Clean and intelligent software designs require a certain organization. Unfortunately, the process itself is not as clear as the end- result. The design process is a long procedure where developers will not be able to avoid mistakes. Some mistakes will be the direct results of inexperience with the type of applications or to simply ignore alternatives solutions developed by peers. Seeing mistakes earlier in the design process reduce development costs. Later code refactoring has a major impact in both development overheads such delayed projects [Ambler2002]. The whole difficulty with design architecture is that the differences between good and poor approaches tend to be subtle. From an external view, it is not always possible to declare one approach better than another one without further investigations. Moreover, different developers could come up with similar solutions that rely on two different paths. Similar to any development phase, the design process is never reaching a final status. Thus, developers need to schedule their time constraints by finding the proper balance between the different phases (design, implementation, testing, and deployment). Designing software for 3DRTS oblige developers to concentrate on priorities that will influence directly the development process as a whole. The set of requirements include understanding the targeted platform architectures and their particularities, which include intensive knowledge on their inner coprocessors, memory size, and bandwidths. Systems architectures need also to minimize both developments time while ensuring real-time performance. Systems architects have to consider many factors such as the system expected lifetime, total development time and costs. They will design different approaches based on the importance of these requirements. Software architectures for mission-critical applications are a mixture of technology constraints and more generic software engineering practices.

Unfortunately, there is no ultimate solution that could emerge from a genius brain but there are definitively some methodologies that can be applied for 3DRTS applications software that resist better to time and technology constraints. This can be made possible by designing a software architecture that repulses basics technology constraints from the global methodology. Surely, the system will still be closely related to the technological constraints that affect the project. However, keeping the control on the side effects of technology is a key element that will define the success and failure of systems architectures and lifetime expectations. This is more important than the inner methodology or technology use within the system. This process is non-deterministic. Since, there are no golden methodologies or heuristics that can be applied, experience, and good usage of well-known techniques such as design patterns are central. This is particularly true with mission-critical software, which adopts ideas that will eventually perform well at a given period but may fail for the next project. This is unfortunate but no tool is right for designing all software.

Software architectures have slowly begun to be more widely used and accepted for creating massive scale 3DRTS software. Even so, they remain an emergent field. The current costs involved in such development as well as the experience learned by developers slowly evolve this attitude to find and adopt better engineering practices including design review and formal discussions on methodology. The needs of better design models for such applications are closely related to the involved complexity [Brooks Jr1995]. 3DRTS result from

combining many components such as a 3D renderer and a physics engine. This implies lot of knowledge from different fields. Much information has to be hidden or encapsulated to reduce the interactions and increase the system readability. For instance, the 3D renderer could come with a set of routines that will accelerate the rendering pipeline by adopting low-level methodologies. One typical example is the OpenSceneGraph toolkit [Osfield] that provides high-level interactions with 3D objects through a scene graph representation but provide low-level mechanisms such as states sorting, culling algorithms for enabling high rendering performance. From this point a view, that information and the methodology use to compute those datasets matters.

At the same time, developers creating high-level behavioral scripts expect to interact with the renderer so it displays geometry on screen but do not need to understand and see the inner techniques [Brooks Jr1995]. Thus, the scale of the software obliges to extract details for specific modules among their needs while hiding most of the complexity by identifying the interconnections and dependencies. The importance of being able to control the complexity is critical in the success of software architectures systems. Many projects fail because they require an important amount of knowledge or because the dependency graph reduces the opportunity to refactor time-critical elements. When a project is reaching this point, it become more and more complex to add or modify the behaviors of single element. This is one of the principal reasons resulting of the abandon of existing systems. Sometime, it may be directly related to some technical problematic, but usually, the growth in complexity showcase design limits that were not visible with reduced complexity.

[Dijkstra1989] has pointed that computer software developments is the only profession where the magnitude of performance increase in few years can reach the magnitude of nine orders and more. He compares those numbers to other professions, and demonstrates that the gap is without parallel. The correlation is that software become more and more complex since their beginning with a magnitude order up to 1^{25} in some areas [Moore2002]. This obliges to analyze software developments differently, not only based on a timestamp, but also to adopt approaches that are fewer dependant on specific requirements. The goal is to reduce the time to rethink software developments every time a new technology appears. For keeping control in this evolution, it is important that the software is subdivided into small subsystems. Humans are generally better to acquired knowledge in small pieces of information rather than in one complicated piece. Breaking, software components means to understand the dependencies that affect some components with each other. This also allows focusing on specific element at a given time. By emphasis, the separation of software into self-contains modules or packages, the component-based approach help to decouple complete systems. Components offer higher level of aggregation reducing the size complexity, so that it remains understandable by programmers. This also prevents the distraction of observing all the low-levels libraries from a higher perspective due to better abstraction layers. This help considerably developers to work within the system and reduce the risks that non experts interfere with some modules, but still provide the tools for specialists to developed state of the arts low-level code. Finally, this separation reduces the introduction of errors by non-field specialists.

4.1.1 The Development Process

Development process can be separate into different phases. The pre-study phase is use to identify the goals and requirements according to the targeted simulations and audiences. It will also help to establish the functional and nonfunctional requirements as well as defining use-cases. This leads to describe systems using pre-study documentations and requirements analysis. The second phase concerns the analysis from a conceptual model of the domain. The purpose is to identify the concept and their associations and define the system functionalities. Ideally, conceptual model and system sequence diagram and dependency graph should be explicitly exposed. The third phase in software design development consists to specify classes' hierarchy and classes' interactions models with the systems, by developing class and component diagram. The concrete implementation is where the ideas are implementing into code. After the implementation phase follow the testing and profiling phases, where the code is tuned for both better performance and more stable code. Finally, the last phase is the software deployment, where the software is finally in customers hands. Often, active support development may be necessary.

4.1.2 Framework Development Complexity

Depending on the system usage, the inner complexity can change. [Brooks Jr1995] describes scaling factor affecting software developments as illustrated in Figure 4.1. He describes that if developing a program for specific and personal use represent a level-1 difficulty for a considered domain, adapting the same program either for general use or extend the program functionalities can be considered as three times more complex to handle. Finally, Brooks Jr. describes that the difference of complexity for developing the same application for

specific and personal use or for generic and general use is approximately 10^2 more difficult. The underlined reasons are the extra work required to polish the application and improve the framework SDK for better flexibility.

		Framework Development Complexity	
		personal use	general use
specific	specific	program (1x)	program product (3x)
	generic	program system (3x)	program system product (9x)

Figure 4.1: Relative software development complexity as presented in The Mythical Man-Month book.

4.1.3 Software Architecture Layers

During the design phase of any software components or libraries, it is important to consider and define the goals and system scopes carefully. [Gamma1995] describes how software architecture can be defined as incremental layers of abstraction and design complexity:

- *Toolkits*: a set of related and reusable classes designed to provide useful, general-purpose functionalities (STL, OpenGL, Direct3D, Boost...)
- *Frameworks*: a set of cooperative classes that make up a reusable design for a specific class of software (MFC, Java Swing, GLUT, Ogre3D...). Many current 3DRTS systems are indeed typically frameworks.
- *Application Programs*: computer programs with a specific purpose geared toward dedicated end-users (Microsoft Word, Microsoft Excel, computer games...)
- *Solution Provider*: set of tools and software solutions that cover three synergistic areas: content creation, production processes, and run-time functionalities [Malakoff2004] (RenderWare, Unreal Technology, VHD++...).

4.1.4 Non Stable Hardware Problematic

One problematic when developing any software applications is to choose the targeted platforms. This is particularly true with 3DRTS, which rely on highly customized hardware, which are constantly improved (see Appendix C). Even the PC platforms, which intend to standardize computer hardware, the technology advances force the developers to adopt new APIs or integrate new functionalities. Thus, for ensuring a good system lifetime, the architecture design need to be adaptable to various platforms that currently exist but also to look carefully of future hardware technology and its impact in the potential system evolution. These varieties in technology oblige to use tools and programming languages that are commonly available. For instance, languages such as Java or .NET are not widely available for 3DRTS platforms. The principal languages are the ANSI C and C++, which became industry standard. The reasons are that compilers for both languages exist on almost all platforms and due to their efficiency, given that they have access to low-level routines and OS features. If the community slowly adopts the OOP paradigm, many developers still do not rely on the fundamental concept behind OOP methodology. Therefore, the CBD paradigm, which is build on top of OOP, is still use in very limited environments.

4.1.5 The Non-Design Oriented Approach

Developing systems architectures for mission-critical software naturally repose on the disposal hardware on top of which applications run. The system designers have to consider the limits and capabilities of targeted platforms to design its architecture accordingly. It is unfortunately common to observe that many interactive simulations are driven entirely by the technology. Newest technology is good but creating systems architectures for massive scale applications that focus on technology is a very risky process. If the focus is toward graphics rendering, the system balance may concentrate on that aspect and trying to run the application on non-graphics intensive platforms may become difficult, when it is simply not feasible. Generally, technical driven software is not very flexible. The risks of on the fly developments require considerations. Most of them become monolithic architectures, where all the code is connected due to weak design. The characteristics of such systems include the tight coupling of data and classes with a high-level of interdependencies that affect the potential to parallelize the code. Another characteristic of bad code cohesion appear when responsibilities for each module is not clearly defined. For instance, this may affect resources management, where it is unclear, which module should release data. All these elements clearly goes against the basic design principles exerts by [Gold2004, McConnell2004], which is detrimental to the system reusability and extendibility. Systems architectures that use negative design such as antipatterns [Brown1999] are less capable of providing generic and reusable solutions to well-known problems. Specifically, the Blob antipattern appears in monolithic code. The Blob antipattern is a procedural-style design leads to one object with numerous responsibilities and most other objects only holding data. A common symptom of this is a single class that seems to perform all possible tasks.

4.1.6 Misconceptions and Solutions to Prevent Them

Ineffective architecture designs are generally the results of few common parameters:

- The solution cover by the system architecture is too complex in comparison to the considered problem.
- The solution is not correct or cannot fulfill the requirements accordingly.
- The development team members do not follow the rules or fail to adopt the design methodology, breaking the overall concept.
- The system subdivisions into smaller components are not enough to keep the essential complexity at a low level.
- The design fails in accepting an incremental complexity.
- The dependency graph does not allow smooth code refactoring.

Almost all the wide literature on programming design books depicts that the only methodology to be applied in such occurrence is a complete refactory. Refactoring such monolithic systems require rewriting large amounts of code to reorder the responsibilities and system workflow. Preventing such situations upfront during the design process is possible by focusing on minimizing both the dependencies between external classes and by dividing the system into self-contained modules. Some characteristics can be proposed for defining the particularities of software design. The primary characteristic is confined to reduce the complexity of the whole system. Good systems architectures facilitate the overall understanding; even so, their internal components may be complex. A second aspect is directly related to the complexity of system maintenance. Clever design approaches may perform better, but the ease of maintenance of self-explanatory architecture is a great benefit for systems lifetime. Another element directly involved with generic systems architectures is their ability to integrate many components with few interconnections. This can be made possible only if components connections remain minimal. Efficient usage of abstractions and encapsulations layers hiding low-levels details help to provide fewer interconnections. This is important for reducing the work of all the development aspects such as integration, maintenance, and validation. When considering the maintenance problematic, many factors have to be remembered. Over the time, many extra codes will be developed and reviewed. Some older pieces may become deprecated or are not able to handle recent functionalities. One might think that it is a relatively easy process to remove older components. However, the scale of such systems and the number of projects that rely on them may prevent such approach. Sometime, duplicated code have to remain within the system for keeping backward-compatibility functionalities, even so it decrease the code readability and increase the overall complexity. In fact, the maintenance and lifetime aspects are a combination of both technical and managing issues [Etherton2004]. When systems architectures are widely used by clients' developers, the need to keep backward-compatibility for depreciated components is primordial. As an outcome, the system management becomes more difficult to handle.

4.1.7 Enforcing Standards and Conventions

Creating systems architectures requires the usage of conventions that affects every development phase. Conventions are the keys elements that should not be underestimate. For instance, enforcing strict coding styles provide a better cohesion [Vlissides1996, Karadimitriou2001]. Coding styles may indicate the ways classes, methods, and fields are named or the way comments and loops indentation are formulated [Sutter2004]. In our system, we use a coding style that combine both Doxygen [van Heesch] tags for comments, while naming conventions for classes and fields names follow the Hungarian notation [Simonyi1999]. These conventions ease understanding code sections. The Hungarian notation prefixes variables names with their types. This helps to distinguish pointers from variables immediately. The Figure 4.2 illustrates the coding conventions use in our framework.

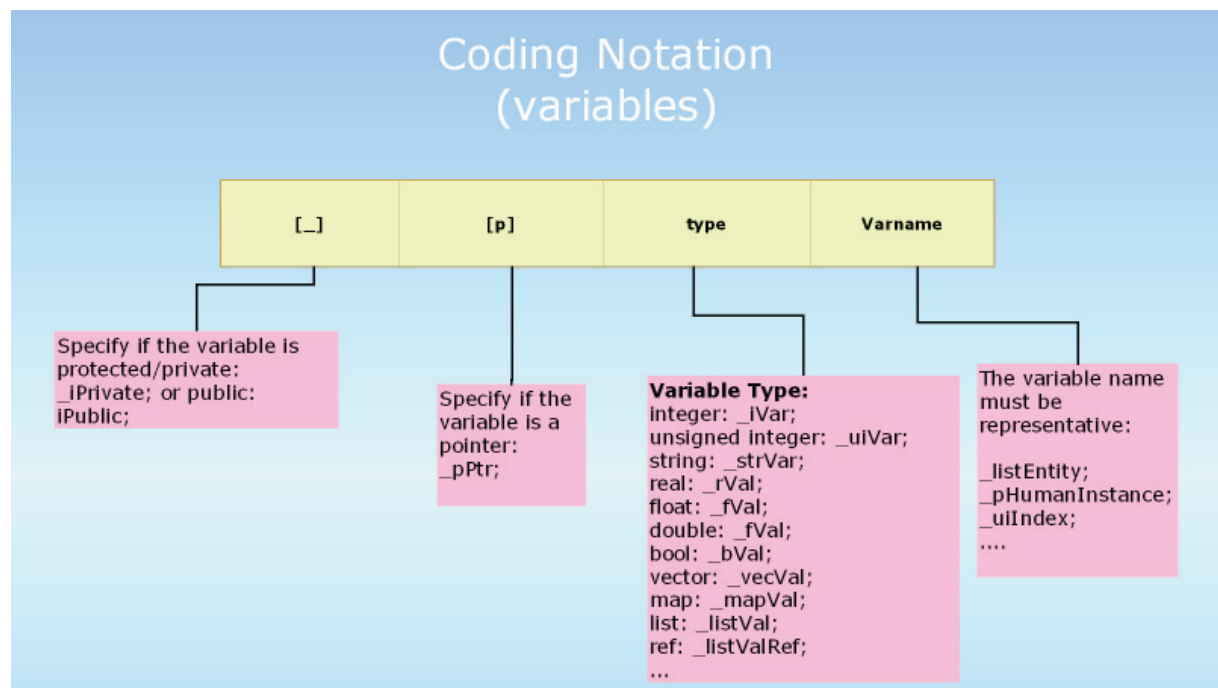


Figure 4.2: Coding style.

These conventions prevent confusion resulting of different coding styles usage. If on the lowest level of the hierarchy, non-standardized codes can be acceptable, high-level components need to follow the rules. Nevertheless, conventions go beyond the simple fact of making the code more readable. They may prevent bad programming practices such as forgetting specifying methods and variables as constants, which may be used by compilers to produce code that is more efficient. In addition, if the system design enforce the usage of patterns such as shared pointers [Boost], it became more protected against hazards such as memory leaks. In addition, the conventions improve code review sessions where peer of programmers can analyze and improve code sections, by keeping a good coherency between modules. Thus, experts are more able to inspect code sections, preventing to some extents introducing programming errors. The only limits with enforcing conventions are when the conventions rules become so complex to be completely mastered.

4.1.8 Documentation

Software engineers tend to reduce the importance of documenting their work. They argue that documenting code is not an interesting task and is useless as other programmers will be able to understand their code immediately. This fact is naturally wrong. Documenting the code remains one of the most important tasks for software engineers and goes beyond writing reports or describing techniques in paperwork. Documenting the code also offer a better probability that peers will master the inner design as well as understanding its limitations and requirements. This is particularly important for systems architects, who are writing reusable code that have to be understood and use by many client programmers. Uncommented projects are more likely to fail or to have a limited lifetime. From a system architecture perspective, the system documentation has to clearly highlight the set of features to be implementing to use and extend the system. For instance, CBD presents a set of functionalities allowing creating additional components with minimal efforts. The documentation should also

exert the system model, including some examples or high-level concepts and their associations. The following listing illustrates such Doxygen comments.

Listing Doxygen Comments:

```
/**
 * @brief Example of XML snippet for the vhdPythonService
 * @ingroup vhdPythonService
 * @author Sebastien Schertenleib
 * @version 0.0.3
 * @code
 * <vhdPythonServiceProperty name = "pythonService">
 *   <bPrintRedirection> TRUE </bPrintRedirection>
 *   <bSynchronizedPythonWithCPlusPlus> TRUE
 *   </bSynchronizedPythonWithCPlusPlus>
 *   <bSynchronizedPythonWithCPlusPlusButRunningInParallel> TRUE
 *   </bSynchronizedPythonWithCPlusPlusButRunningInParallel>
 *   <bGarbageCollectorEnabled> FALSE </bGarbageCollectorEnabled>
 *   <iPythonThreadPriority> MIN_PRIORITY </iPythonThreadPriority>
 *   <strInitializationScript> .\initializationScript.py
 *   </strInitializationScript>
 * </vhdPythonServiceProperty>
 * @endcode
 * @example vhdPythonService::XML-Config
 */
```

To understand the interaction and association of classes within the systems, the ability to extend classes and interactions diagrams give a better picture to the client programmers. Finally, the components and library APIs need to depict their different parameters and provide the information on pre-and-post conditions notably for handling exceptions and code errors.

4.2 Architecture

A great deal of the technical innovations in computer technology is spurred by the desired to exploit and improve 3DRTS, both in hardware and software. The investments in R&D for 3DRTS is now reaching billions of dollars every year, where single project budget commonly reaching 8-digit figure [Swartz2004, de Gentile-Williams2005]. In fact, PC hardware innovations from CPUs to graphics and sound chipsets are clearly connected to 3DRTS. This constant improvement is also affecting the PC market. Customers need to constantly upgrade their computer with newest hardware. The technological race force developers to face another difficulty for reducing the climbing development costs. One critical issue is to deal with scalability. System architectures that are flexible enough to handle well the notion of scalability will have more significant advantages than system architectures with better performance at a given point in history [Stutz2004].

4.2.1 Objectives

In our approach, we want to promote a higher level of flexibility, which keep under control the rising production costs. The idea behind the architecture is to use various technologies while ensuring an appropriate level of flexibility handling a wide range of interactive simulations domains. The restricted resources to develop the system architecture oblige to choose priorities. Thus, the focus is made on the architecture rather than on pure RAW performance. The system architecture qualities do not rely on its ability to draw more polygons or virtual humans than the competitors do but to provide a set of patterns and coherent APIs, which give more freedom to the developers. In effect, The RenderWare framework [Criterion] had already proven that a solution that provide better environments can prevail on more advanced software technologies, which may be unproven or too restrictive. Surely, performance is not ignored but comes with a lower priority. The Table 4.1 shows the repartition of priorities for our architecture and standard game engines.

Table 4.1: Software development priorities.

Element	VHD++	Game Engine
<i>Flexibility</i>	3	4
<i>Performance</i>	5	1
<i>Coherence</i>	2	6
<i>Portability</i>	4	2
<i>Reliability</i>	7	7
<i>Maintainability</i>	8	8
<i>Pipeline</i>	6	3
<i>Architecture</i> (concurrent, patterns)	1	5

The repartition of the quality attributes differs, as the main goals are also different. Game engines focus on set of specific time-based requirements to maximize the resources usage at a given point. The reuse and code coherence is not as important as in our case [Fristorm2004]. They need to work on elements that will influence directly their final products. If such systems do not need to provide the abilities to work with different simulations domains, the design requirements and methodologies can be considerably reduce. This goes against the current proposal, which intent to verify and validate architectures designs methodologies less dependent on specific technologies. In addition, the scope and the period disposal in developing this architecture do not allow to fully optimizing all simulations elements. The difference of resources involves in this project in comparison with commercial solutions such as [Criterion] or [Epic-Unreal] is an order of magnitude of 10^2 for both humans' resources and investments [de Gentile-Williams2005]. Thus, in the purpose of this thesis, the analysis method focuses on providing solutions for concurrent systems architectures. To validate the ability to support quality attributes for high performance, dedicated components using both low-level to high-level representations were developed related to multi-agents and crowd simulations. The idea was to demonstrate that high-level and well-defined systems architectures could perform efficiently in real-time.

4.2.2 Build for People

Building software architecture, which can cover different simulations domains differ from single application systems. The code and architecture has to be readable by humans. This implied that in some circumstances, the code might not look and perform into the most optimized form. However, ensuring a descent readability is the key element for understanding the system, which involve better proportion to evolve and reduce the development time with better errors rate [McConnell2004]. The process of writing readable code is not necessary inducing longer time to write the code. The emphasis has to be applying on being consistent over the project by adopting coding styles that are relevant. Moreover, describing algorithms and APIs with meaningful comments also help clients' programmers to understand the input and output parameters as well as side effects. This is important as otherwise; clients' programmers may waste time debugging low-level routines due to a wrong API usage. In addition, a common behavior with programmers is that they tend to generate code for themselves with limited efforts, as they think they will be the only person working on this set of instructions. Professional programmers should avoid this very dangerous practice [Read2003]. In fact, [Parikh1983] have analyzed that code maintainers can spend up to 60% percent of their time only to understand the code they have to handle, mainly due to poor documentation.

4.2.3 Hardware Scalability

With development time duration of 18 to 36 months, the problematic to provide the adequate assets complexity for the product is difficult. If the system cannot provide any scalability with end-users platforms, it may results that the simulation will not be able to compete with current simulations. Moreover, the diversity in abilities of end-users platforms requires being able to scale the simulation to different hardware. One of the principal reasons is to growth the potential user base, by providing the simulations for many systems. The same behavior affects PCs-only projects, as PCs are widely varying in performance. End-users expect that 3DRTS perform well on older systems, while using the latest technologies whenever they are available. The challenges face by the developers requires designing systems, which take advantages of techniques and algorithms that scale content on demands.

4.2.4 Software Scalability

The creation of scalable software can be separate into different categories. The first category is to use off-line methods for exporting assets base on the platform abilities. The same model will be exported with different geometry complexity and textures quality. This allows adapting the system when a platform does not handle some data like compressed textures for instance. Techniques and algorithms that will adapt the model complexity on the fly compose the second category. This allow the simulation to be more flexible base on the current workflow but require slightly more processing power than offline methods. The good usage of both offline and online methods allow the simulation to not rely too heavily on the base platform.

4.2.4.1 Scaling the Environment

When developing a scalable simulation, it may be difficult to do all the verifications if the interactions differ from platform to platform. One methodology is to work with a base platform in mind. All the interactions have to work well on this platform. Then when additional processing power is available, the simulation will only scale up areas which do not alternate the interaction models [Pallister1999]. To illustrate this technique, let us analyze the following example. Imagine that the application intend to simulate a city fill with a crowd. The crowd itself is separate into different categories of characters. Some characters can interact with the user, while others only serve as ambient characters. When running on a better hardware, adding interactive characters would change the interactions model requiring the development team to validate the simulation for each platform. On the other hand, adding ambient characters would not interfere with the simulation but the system will still benefit from better hardware. In fact, the video game “*Republic: the Revolution*” from Exilir Studios was keeping the frame rate at a constant refresh rate of 20FPS. Depending of processing power the city was populated with more cars and virtual humans, which were not directly affecting end-users experiences. Other similar examples are to include richer features designed to run on higher performance systems such are animated plants, clouds simulations or post-processing image filtering. By adding those elements, the developers ensure that the system will react identically on any platform. Ambient objects or characters have been already exploited in simulations like Unreal [EPIC-UnrealGame1998] which feature flying creatures or such as the maintenance robots in the game Jedi Knight [Lucasarts2003].

4.2.4.2 Scaling 3D Models

When creating 3D models, artists can create highly detailed geometry and textures. For keeping interactive frame rate, models have to be scale down reducing their LOD and by compressing their textures [Bjorke2005] as shown in Figure 4.3. One problematic is that the artists will be responsible to adapt the model for each platform. Semi-automatic methods may help in the process, but the artists still need to check them out. This process may be long and the different LODs may create visual glitches when switching from one level to another [de Heras-VSMM2005]. Progressive LODs provide smoother results at a computing cost. Those techniques include the usage of parametric curves and surfaces and are perfect candidates for being sent to the GPU with very low bandwidth requirements. Parametric surfaces can really help in making a 3DRTS scalable, but using this approach requires a paradigm shift for developers (including both the artists and the programmers) [Boyd2003, Luebke2003].



Figure 4.3: Mipmap allow optimizing the texturing performance by reducing the memory footprint required for rendering distant objects.

Nowadays, most artists and teams continue relying purely on polygons meshes. Thus, scaling 3D geometry involve to alter the original model by removing polygons and textures quality to maintain an interactive frame rate. Taking the problematic from the other extreme, a technique for increasing the complexity of a model on the fly was developed. This method refers as subdivision surfaces intend to subdivide the mesh in sub-meshes, improving the object precision [Akenine-Moller2002]. Imagine that you want to render a circle and that the model comes with only a few points forming more a square than a circle. If you subdivide the mesh into 2, 4 ... you will obtain a much better approximation. As the subdivision is made in the GPU, no extra data are sent to the GPU. This technique may be used in all CPU bound applications (see Figure 4.4).

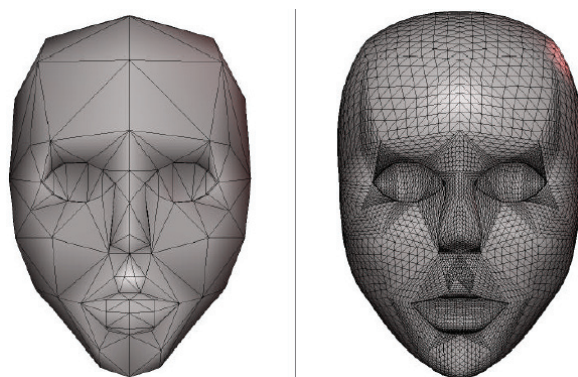


Figure 4.4: Increasing 3D mesh geometry using subdivision surface methodology.

4.2.4.3 Scaling Character Animation

In the early age of 3DRTS, characters animations systems were using pre-stored animations to move the characters and objects in a scene. An artist had to record all the different poses manually. The idea was to reduce the workflow at run-time. With current hardware, more realistic and dynamic techniques like skeleton animations become possible [Adams2003]. They give more flexibility allowing reducing the number of bones that influence a vertex position and orientation. For instance, when a character is faraway from the camera viewpoint, the animation engine can ignore computing the entire skeleton every frame [de Heras-VSMM2005].

4.2.4.4 Scaling Special Effects

Adding special effects to 3DRTS is a way of improving the rendering quality without varying the core experience. This allows spending extra CPU cycles without altering the interaction models. Therefore scaling special effects is easier to implement. Special effects like particle systems can scale effectively and can be switch off on slower hardware. Resourceful developers are also investigating methods that are more sophisticated. Other techniques improving the rendering quality are likely to be adapted on request. Some of theses techniques include anti-aliasing or anisotropic filtering or additional pixel shaders' effects. The Figure 4.5 depict some additional special effects used in the video games "TOCA 2" published by Codemasters. Those effects are in placed only when enough processing power is available [Perminov2004, Davies2005].



Figure 4.5: The image on the left show the game running a single core processor, while the image on the right run on a dual core CPU, where one core is use for eye candy special effects [Davies2005].

4.2.4.5 Scaling Methodology

Every system applications have its own capabilities for supporting scalability. Some interactive applications genders are better candidate for providing high level of scalability. Systems architectures will likely provide different techniques from off-line to on-line handling. Regardless of the inner capabilities, the difficulty remains of finding the proper balancing when scaling the information. Does the simulation suffer more with restricted graphics but realistic physical based interactions or do the graphics need to be maximized to increase the character personality? When choosing the quality and variety to be used, many approaches have to be considered. The users to fit its own expectations can configure the system. Some users may prefer to sacrifice quality in the rendering pipeline for a smoother frame rate while others would prefer clear textures and higher resolutions [Walrath1999]. If this level of interaction is possible on PCs, generally embedded platforms do not allow such configurations. Developers can decide to set some constraints base on the hardware limitations that they can analyze by computing pre-deployment runtime profiles that serve as benchmarks. Thus, assuming the simulation should always ensure a minimal frame rate; different configurations can be stressed and observed. Therefore, developers can choose the optimal configuration that combine good quality and excellent performance. However, such a benchmark would not be representative of the whole application. It is unfeasible to create a generic benchmark. The real-time performance will differ from scene to scene. Thus, a third method is to embed runtime profilers. The idea is to analyze the system workflow and to dispatch tasks and CPU time to the different components. Therefore, when at a specific time, the scene is heavy in geometry; the system can balance the workflow accordingly. This methodology is also practical for controlling agent's AI. For instance, crowd simulations are quite intensive in processing the agent's behaviors [Reynolds2006]. Thus, being able to reduce and balance the workflow in computing behaviors is important. Therefore, the system may give more CPU resources to the AI component base on the scene geometrical complexity. Without such mechanisms, the experience will suffer of inconsistent frame rate, affecting the sensation of presence. In our architecture, the system scalability relies on those different areas providing automate scheduling of tasks, while still offering several options to end-users.

4.2.5 Adding Variety

Creating simulations with rich environments require producing many assets. However, there is a need to reuse information whenever it is possible. For instance, by creating objects or characters that shared the same geometry with different textures or shaders increase the abilities to create virtual worlds that are populated with unique individuals [Carucci2005]. Also adopting methods for procedurally generated objects is necessary for improving the content variety [Katchabaw2004, Tatarchuk2005].

4.2.5.1 Variety in Textures

The creation of 3D objects such as virtual characters is a complex task. In earlier 3DRTDS, variety was restrained to textures modifications. For instance, to change the color of clothes, a new texture with the dress color was made as seen in Figure 4.6.



Figure 4.6: Variety in textures.

In this example, we can notice that the dress stays identical, except for the colors. We would like to color the belt in one way and the toga in another. To colorize, characters use a shared texture that carry the gray scaling lighting information as can be seen in Figure 4.7 on the left. In [de Heras-VSMM2005], we describe a

technique for adding colour variety. This technique allows preserving the skin modulation as a separate process due to the particularities of human skin colours.

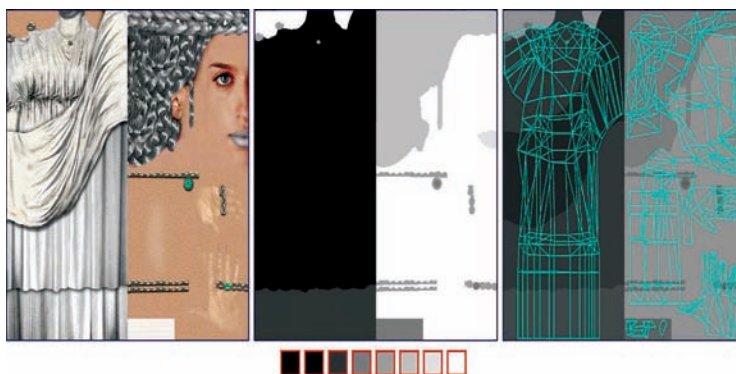


Figure 4.7: The base variety texture displayed along the alpha map in the middle and the triangle laid out in the right. The square beneath the alpha map show the alpha colors.

Using this technique, designers can create near to infinite distinct individuals that shared the same geometry and textures information. These free artists to generate multiple instances of the same objects and improve the believability when making crowds. The Figure 4.8 depicts some examples of concrete instance made from the same dataset.



Figure 4.8: Base variety texture (most left) and the alpha map (next to it) with some combination of colors.

4.2.5.2 Variety in Behaviors

The variety in behavior combines higher-level behaviors such as representing virtual characters emotions and through low-level motion systems. [Brogan2000] describe simulation LODs controlling locomotion variety. [Thalmann-SIGG2004] goes further and demonstrates that low-level controllers can increase the believability of a crowd by reducing the uniformity. [Sung2004] propose an approach for controlling the scalability and variety in crowd behaviors. They separate virtual humans into categories where basic agents have limited behaviors while situation-specific agents are equipped with behaviors that are more detailed. Their system can also promote and demote agents from categories upon simulations events. However, even so techniques for generating procedurally virtual humans' behaviors exist; most variety in behaviors continues to be created manually by designers.

4.2.6 Reusability

Software reusability is an attribute that refers to not only improves productivity but also has a positive impact on the quality and maintainability of software products. Software reuse has been cited as the most effective means for improvements of productivity in software development projects [Boehm1984, Paul1995]. Reusability is at the center of CBD. CBD evolve from previous design and programming paradigms [Szyperki2002]. CBD is both a subset as well as an extension of current software engineering practices with a good emphasis on component reuse. CBD allow managing multiple implementations of the same component in different technologies as they evolve with time. In addition, it gives better opportunities to validate components behaviors. The move toward reuse is becoming so widespread that it has even changed software industry's

vocabulary. For example, software acquired externally is described as *commercial-off-the-shelf (COTS)* or third-party software, *commercially available software (CAS)*, and *non-developmental item (NDI)*. When use as parts of systems, they are called components. The systems themselves are known as component-based software or systems of systems [Voas1998]. This trend can be explained with the continuous rising costs of software developments, especially for 3DRTS. In fact, the costs to develop a new system from scratch are significant. This renders custom software developments very expensive. Embedding large-grain software components into multiple applications spread the components developments costs.

4.2.6.1 Software Reusability Validation

Software reuses oblige developers to maintain the existing code base. For being benefits, the cost of software reuse should remain smaller than the cost of developing new components. The amount of work needed to reuse a component in another system of the same domain may be determined on measurements described in [Mao1998, Itkonen2003]. [Caldiera1991, Dandashi2001] propose a model that describes three reusability factors:

- Cost of reuse.
- Usefulness of reusable components.
- Quality of the reusable components.

Analysis for observing the benefits of software reuse can be conducted using well-established economic techniques. Many criteria enter in the equation, notably cost-benefits analysis as seen in **Equation 4.1** [Bin1998]:

$$CostSaving = CostScratch - CostReuseOverhead - TotalActualSoftCost \quad (4.1)$$

The cost saving can be determined by applying one of the many cost estimation models. The overhead costs associated include:

- Domain analysis.
- Increase documentation to facilitate reuse.
- Maintenance and enhancements of reuse artifacts (documents and components) [Johnson1988].
- Royalties and licenses for externally acquired components.
- Creation (or acquisition) and operation of a reuse repository.
- Training of personnel in design for reuse and design with reuse.

The actual cost of the software will include project-related reuse costs, such as the adaptation and integration of reuse artifacts. The *Net Present Value (NPV)* technique is a well-established and popular method for conducting a cost-benefit analysis [Bin1998]. The *NPV* method determines the present value of a stream of cash flows that result from creating a component or establishing a reuse program. The technique determines the attractiveness of reuse as an investment compared with other software development strategies. Here, costs refer to total life cycle costs, or those incurred from investigating, designing, coding, testing, debugging, and maintaining the components. Similarly, benefits include costs saved. Benefits also include additional profits resulting from earlier products completion. The benefits associate with system reuse can be expressed as a ratio as depict in Equation 4.2 [Bin1998]:

$$Ratio = (CnoReuse - Creuse) / CnoReuse \quad (4.2)$$

4.3 Conclusion

In this chapter, we have presented the main design principles and scaling factors that oblige software architectures to deal with many problematic from design styles and naming conventions to content variety and memory management. In the next chapter, the discussion will describe how CBD can be extended with data-driven mechanisms using multi-view representations. This will lead to analyze the concurrency model of our system, which optimize resources usage and system performance.

Chapter 5

Component-Based Software

This chapter introduces the overall architecture design with an emphasis on CBD and Interactive Simulation Objects (ISOs) coupling. Notably, we will analyze how design patterns can be adapted for mission-critical software. The discussion will also describe the problematic appearing with concurrency model such as data synchronization and data flow.

5.1 Component Model

When developing 3DRTS, we need to distinguish different set of features that form the system. A sum of that covers the compulsory contributions such as a 3D renderer or animation controller. The second category is modules that extend the system with unique features (see Figure 5.1). These include particular rendering techniques, flexible AI management or scripting interfaces. Finally, the last set of functionalities is features that use for polishing the applications. They are not mandatory for enjoying the experience but they come under the shape of eye candy visuals or additional environments. With projects and hardware architectures becoming larger and more complex, this clear separation is necessary for ensuring that core elements can be reusable. Development a platform with high reusability potential is still not widely use, as they appear more complex to design. Code reuse is classified into the following categories [Gill2001]:

- *Copy-Paste reuse*: duplicate same code into different source files.
- *Functional reuse*: cover set of functionalities that are use across many components. Those subroutines define set of utility functionalities that are place in a function so it can be call multiple times. This decrease the maintainability issue of duplicate source code, while allowing placing platform specific functionalities into separate source files.
- *Horizontal reuse*: represent stand-alone library use within different projects. Often, the horizontal code reuse may have a limited lifetime as their developments is separate from the core system.
- *Vertical reuse*: represent large code base use for current and future products. Such components are designed with reusability in mind, which makes them more future proofs than previous approaches. However, as the design is slightly more complex since these components need to be capable of growing.
- *Components reuse*: defines set of functionalities that can communicate with other components from higher-level perspectives. Such components can evolved independently and ensure that they can be reuse across projects.
- *System reuse*: defines systems architectures that take advantages of the different reuse methodologies for building better reusable systems. They need a clear architecture design from the beginnings that allow the system to be extended in a flexible way.

One might argue that software development for 3DRTS evolves too fast for allowing developing systems architectures with dedicated attention for code reuse. If some low-level modules relying on particular technologies are likely to changes from project to project, there is an important amount of code base not directly affected by end-users platforms. Even low-level modules like 3D rendering continue to perform efficiently principally with polygon-based meshes. Therefore, the principal question is not to know if developing systems architectures with a good proportion of code reusability is desirable but rather to consider the benefits of this method. For one problematic, there are multiple possible implementations and algorithms. However, some functionality can be seen as atomic or straightforward routines. For instance, developing a hash map may be seen as a simple task. However, the likelihoods of creating such routines without creating bugs are very low. Thus, being able to rely on well-proofed mechanisms provide an important productivity boost as well as offering a more economic process. Reusable code developments require some strict attitudes and skills. The programmers cannot provide quick improvised implementations that will work only for dedicated purposes. One of the difficulties is to be able of specifying right from the start of the development cycle, which modules are intended to be reusable. Clearly, some components or functionalities will have a shorter lifetime. The architecture design will have to translate the mechanisms that have a good potential of being reuses for future

applications and hardware. This fundamental design process will be responsible of defining the major components and vertical reuse modules. Horizontal and functional reuses are less dependent from the design and provide to the programmers the ability of reacting on pure prospective implementations and problematic. On this level, developers could adopt the *eXtreme Programming (XP)* methodology [Beck2004]. However, the use of *XP* requires good technical skills to reduce the risks of altering working code [Fowler1999]. However, *XP* and reusability can be combining as *XP* promote developing functions and classes that are clearly use, which suited the code reuse characteristics.

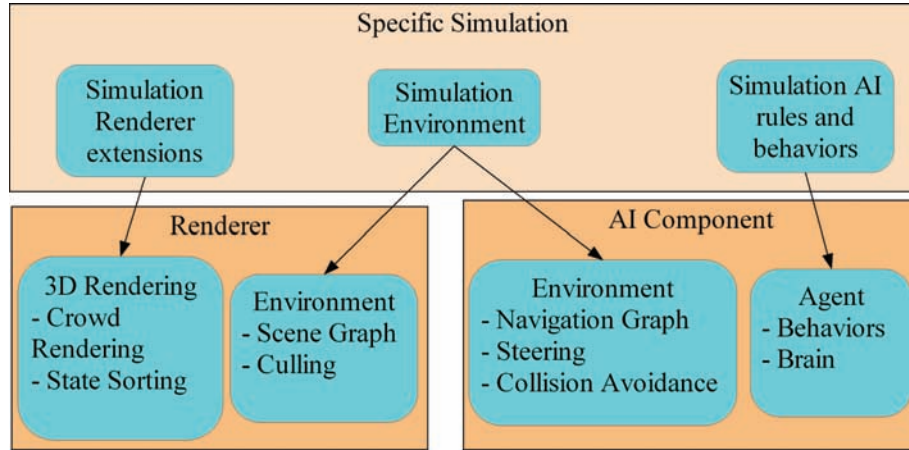


Figure 5.1: Components and the simulation specific layer.

5.1.1 Component Based Architecture

Many developers believe that software reuse for embedded systems are almost impossible. They think that producing reusable and modular software require tremendous experience and knowledge. In fact, it principally require a strong discipline. Trying to master the complexity of systems architecture into a single piece of software is counter-productive. Some parallel can be draw with the mathematics world [Polya1957, Dijkstra1965, Gutek2003], where solving mathematics problems consist to formulate and separate the problematic into smaller known problems. Then the global solution is the combination of all the intermediate results. The goals are not to resolve the problematic with a brute force methodology but to circumvent the difficulty. The analogy to software developments is that going from ideas directly to code may work for some projects; it will fail for massive scale software. It is more accurate to divide the complexity in more reasonable elements that can be developed with known techniques, reducing the time dedicated to debug the system [Telles2001]. A layered architecture offer better potential for code reuse and extensibility, as the coupling between components is minimized and are less connected to the underlined technology. Writing modular software is characterize by promoting simple structures and data encapsulations into coherent abstractions layers, which separate the interfaces from the implementations [Stevens1974, Parnas1985, Schach1993, Sweeney-GDC2006]. The highest level of modularity refers as reconfigurable component, where a component can be interchange without minimal efforts. In non-modular architectures, the module interfaces are directly connected with other modules. In C++, modules that require including other modules become dependent on these interfaces. As an outcome, changes in these interfaces break the behaviors of both modules. In contract, interfaces specifications for reconfigurable components are designed around a predefined standard and are independently from other modules. In this approach, all the interactions between the components occur through these standard interfaces.

5.1.2 Multi-Layers Architecture

The architecture retained in this thesis is subdivided into four different categories (see Figure 4.4). At the bottom, we find the hardware communication layer. Since mission-critical software runs on dedicated hardware, to optimize the system performance, developers rely on low-level libraries that compute datasets efficiently. For 3D graphics, this means to use 3D graphics APIs such as OpenGL and DirectX. With these, the system can exploit advanced features directly supported by the different co-processors. However, these low-level APIs do not provide any organization on the data. Therefore, a higher abstraction level responsible to handle the platform separation and scene organization is needed. Such components refer, as *Hardware-*

Dependant Components (HDC) can be designed on two layers: a common cross-platforms interface and a platform specific implementation. *HD* interface components convert hardware-dependent signals into hardware-independent data, allowing other components to communicate with HDC. Their interface replaces specific hardware calls by a generic layer using either abstract classes or the *pimpl* (private implementation) pattern. A typical example is the 3D rendering APIs abstraction as the one developed in Ogre3D [Streeting]. *HDC* provide similar functionalities than pure generic components, but yield to increase performance, because of hardware specific optimizations. Unlike the interface components, *HDC* directly communicate with hardware. In our model, we choose to use toolkits that handle a unique technology at a time, that is, one toolkit might handle graphics or physics but not both. The underlined reason is to minimize the coupling of components, offering higher potential for replacing an older technology. This is particularly important for 3DRTS that are highly connected to very specialized hardware that evolve constantly. For instance, with the apparition of PPU, it becomes possible to handle much complex physics simulations. However, this obliges to adopt specific APIs. Therefore, by reducing the number of elements that would need to be rewritten, the components can evolve with optimum code reuse.

The next layer represents the core architecture responsible to handle the system workflow that accept input streams from configuration files and scripting languages as well as spreading tasks among the available processors. As this layer expose the functionalities for client programmers, it is important that a consistent vocabulary and stable APIs are deployed. One difficulty consists to find the proper balance in terms of features integrated in the core infrastructure. Finally, the last layer represents application specific elements. For instance, in an augmented reality simulation, there is a need to have a component responsible to capture and track features within camera images. This kind of components is unique for AR applications and should not be part of the core infrastructure. By letting application developers the opportunity to extend the system with their own plug-ins, improve the scope of applications supported by the same architecture. Finally, the application presentation layer in addition to C++ specific components integrates the simulation code written in scripting languages.

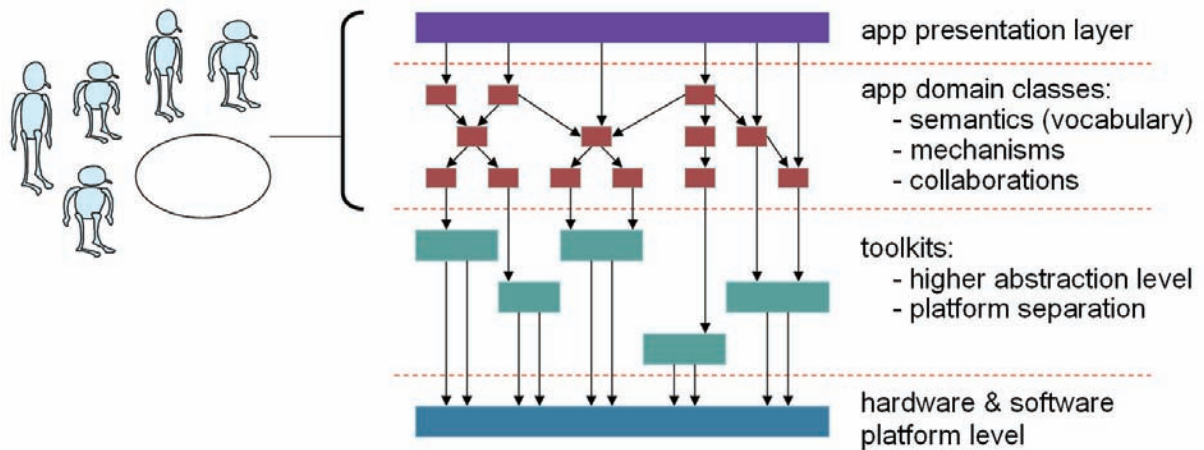


Figure 5.2: Multi-layers separation.

5.1.3 Abstraction

Abstraction is a very important concept for software modularity. In our model, this refers to encapsulating one technology (graphics, physics, sound...) into a self-sandbox component. The purpose is to minimize the interconnections of components. This eases the replacement of one component embedding a technology by a new implementation. This is critical for 3DRTS developments that require being pro-active for adopting new techniques. For instance, the system might employ a physic simulation component based on a software implementation. In addition, with the apparition of PPU hardware, we might consider to develop a hardware-accelerated implementation. By keeping the coupling of components at the higher abstraction level, the system can be more responsive to technological advancements. This is made possible by describing stable APIs that configure set of events. For physics simulations, a function would serve to notify, when an object collide with the environment as describe in the following listing. Then depending on the corresponding implementation, the behaviors would be different but the events themselves remain unchanged.

Listing: Physic interface

```

class IPhysicInterface
{
public:
    void collide( uint uiID1, uint uiID2);
    void setPhysicAnimationControl(bool bFlag);
    void update(vhtReal rElapsedTime);
    void setForce ( uint uiID, const vhdVector3 & vecForce);
    //...
};

```

5.1.4 Loose Coupling

The efficiency of concurrent systems is tightly connected to the ways modules and packages are coupled. For concurrent software, the simpler the connections among modules can be achieved, the greater potential for code sections to run in parallel exists. A better design consist to associate modules instead of attached them directly. This may imply that information has to be duplicated but creating associative components guarantee that they are loosely coupled. The criteria that describe the components coupling is the addition of the following factors:

- *Encapsulation*: a good use of programming model which hide the complexity on the lower level. It uses inheritance, encapsulation, and delegation patterns and reduces the components connectivity. Providing an API, which reduces the interference with the intrinsic functionalities and data make components slightly better for concurrency models.
- *Flexibility*: by keeping the number of connections between components to a low level, the system can also more easily change its behaviors or another component can replace an older one at a low refactoring cost.
- *Flat API*: creating an API which expose few classes and for which each functionalities rely on basic parameters is more important than creating an API that allow to control low-level data manipulations, which increase the indirect coupling of components.

More easily the components can interfere with each other, more loosely coupled they became. Creating a system architecture that minimizes the interconnections is beneficial for both the flexibility and maintainability of the whole system. For being able to design with this perspective in mind, we need to analyze the type of connectivity that may affect modules. A direct connectivity affects the objects themselves. It occurs when one object need to communicate directly with one or a chain of objects. CBD keep the objects coupling within components but are by definition not exposed to the component's API. In the practice, some objects coupling can still affect system elements due to improvised fixes or through programming “*hacks*”. Coupling within a component API is also described as *Components Communication Layers*. At these relatively high-level layers, the communication is mainly done through messaging operations. This reduces the interdependences between modules. Finally, the most critical element is the Object-Parameters coupling. This happens when data are shared across components. Different techniques may restrain the propagation such shared data like duplicating information with a centralized organizer, while other rely purely on data synchronizations mechanisms. Both approaches have their pro and cons. For instance, data duplication increases the memory footprint and obliges the system to ensure that the information reach every element to avoid inconsistent data. The unique dataset approach do not need to wonder about these problems, but increase the potential that threads have to wait on lock acquisitions. Thus, depending on the platform, the best choice may differ. For systems with important amount of memory and highly concurrent hardware architectures, the data duplication overhead is worth due to a mere efficient use of the processing throughput.

5.1.5 Dependency Graph

Data structures such as scene graphs use a hierarchical dependency representation [van der Beek2003]. They reduce the computation of world transformations during a tree traversal, which can be a costly operation. Settings dirty flags in the scene graph usually optimize the calculation. If we have three objects A, B and C, where A is the parent of B then if A move, it will affect automatically the world transformation of B. This example arise some of the issues related to dependency graphs. A second problem that requires attention is cyclic dependencies such as the one presented in Figure 5.3. Cyclic dependencies are best avoided, as they do not perform well in concurrent systems.

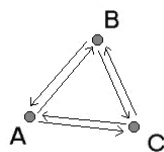


Figure 5.4: Cyclic-Dependencies between A, B and C.

5.1.6 Keeping Localized Dependencies

The natural evolution of massive scale software increases the probability that many components interconnect with each other. At first, it might seem practical to centralize the information. However, this generates interdependence between modules increasing the difficulty to maintain the code base. Moreover, data sharing on concurrent systems waste precious CPU cycles for lock acquisitions and for fetching data into different caches. Designing the system to clearly separate components that do not share any relations is a key factor for code reusability. By localizing the dependency graph on the lower level, reduce the risk of creating unmanageable systems, even so the information may have to be duplicated. The slight memory footprint and performance penalty is the minimal cost involved for developing such system. In addition, even ad-hoc systems may use set of libraries that come with their own mathematics functions requiring converting data anyways.

5.1.7 Indirection

The concept of indirection serves to prevent that the objects keep their behaviors internally. The idea is to create objects keeping internal references on other objects to delegate tasks execution. In this scheme, an object does not directly integrate all the other elements, reducing the object coupling. For instance, the following listing describes an object containing all its behaviors.

Listing: Object encapsulating all its internal behaviors

```

#include <geom.h>
#include <sound.h>
class Object
{
    GeomObj gObj;
    SoundObj sObj;
};
  
```

One drawback with this approach is that the objects are directly connected to specific sub-objects. To interchange the behaviors of any sub-object, the class definition needs to be adapted. To overcome this limitation, the idea is to delegate the implementation. This allows replacing the concrete implementation of any sub-object with no impact on the considered object as described in the listing below.

Listing: Object encapsulating all its internal behaviors

```

class GeomProperty;
class SoundProperty;
class IndirectObject
{
    GeomProperty *pgObj;
    SoundProperty *psObj;
};
  
```

5.1.8 Orthogonality

The concept of orthogonality [Sweeney-GDC2006] refers to situations where two components can run simultaneously without limitations. For instance, the 3D sound and 3D graphics renderers can act on the same objects independently. The idea is that a component can be used without worry about mutual exclusives. These mutual exclusives occur when the use of a feature prevent the use of another component features. In our

architecture, every component runs independently. This minimizes the knowledge required by client programmers, as they do not need to find appropriate compromises in the set of available features.

5.1.9 Object-Oriented Programming Methodology

OOP try to distinguish the data being manipulated from their manipulations. Therefore, instead of observing data as a flow from which external functions are modifying them, OOP encapsulate all the manipulations and its data within a single object. Thus by looking to the objects, it is immediate to see which manipulations can be applied on them and prevent that data are changed when unrequited. OOP offer cleaner designs where developers can think on an object level rather than on functional level. Thus, the way data are manipulated should not depend on the implementation details, which allow changing the internal implementation. Another benefit of OOP comes with the inheritance and polymorphism patterns. Inheritance allows object to inherit from other objects. They can be treated exactly in the same way as if they were those objects, which allow hiding concrete implementations details and layered functionalities. Polymorphism allows extending the manipulations and mechanisms that affect the objects. Thus, the same mechanism of communications can have different behaviors on a per object basis. C programmers have always insisted that C++ and its related OO mechanisms were slower to compute than non-polymorphic functions. Indeed, this assumption is exact. However, we have to consider the real cost and overhead involve with OOP mechanisms as shown in Equation 5.1. Assuming the method does not perform trivial operations and the call frequency remains relatively low, then the flexibility offer by OOP overcomes this minimal overheard impact [Lippman1996].

$$overhead = callFrequency * \frac{functionCallTime}{functionBodyTime} \quad (5.1)$$

[Driesen1996] have shown the related overhead for calling a virtual function is of about 13% as depict in Equation 5.2. They discuss that this overhead could be optimize in the range of 20%, but the nature and mechanisms involved for managing a polymorphic call prevent further optimizations at least with the current set of language and hardware.

$$relativeOverhead = \frac{virtualFunctionCallTime}{nonPolymorphicFunctionCallTime} \cong 13\% \quad (5.2)$$

5.1.10 Productivity

In the early age of 3DRTS developments, the processing throughputs were restraint. Therefore, the priority was to optimize the code and assets for optimal run-time performance. Nowadays computers hardware together with the increased expectations for wider and more complex virtual worlds modifies the balance between performance and productivity. Within the development team, the principal bottleneck is confined to the content creation, where artists need to handle a complex content production pipeline. As an outcome, there is a need to interchange some degrees of performance for a higher flexibility and better productivity. The benefits of building an architecture with the techniques of abstraction, indirection, and orthogonality in mind, offer a more general and extendible framework.

5.1.11 Investigating Existing 3DRTS

For elaborating a common infrastructure, it is important to identify a set of repeating architectural patterns across the different types of 3DRTS. This implies to extract set of repeating mechanisms, services, and components. This includes low-level APIs handling optimized math routines or file systems management. This also affects object manager that handle ISOs (Interactive Simulation Objects) lifetime. Thus, it is critical to organize and categorize the different elements using distinct requirements (see Table 5.1). For instance, the 3D renderer is generally considered as a time-critical element, where code optimizations are critical to ensure we can display high detailed virtual worlds. In the other hand, many other elements are not time-critical and benefit more from a higher flexibility than pure run-time performance. For example, the AI component responsible to handle virtual humans' behaviors requires offering a high level of flexibility. In effect, for improving the believability of the virtual world, designers must fine-tune many parameters. Being able to distinguish these distinct requirements allow to combine both high flexibility and efficient run-time performance.

Table 5.1: Categorizing components.

• time-critical	vs.	non-time-critical
• performance heavy	vs.	light
• synchronous	vs.	asynchronous
• data sharing	vs.	transmitting
• compiled	vs.	interpreted
• compile-time	vs.	run-time configuration

It is important to differentiate the elements that form the architecture between the core infrastructure and the simulation code. If the core C++ code is shared between the applications, the simulation code is specific and subdivided into two categories. The first category covers extensions written in C++, where specializations of basic classes are developed. The second category affects simulation code that is not time-critical like simulation events and virtual humans' behaviors that is developed using high-level scripting languages, easing the prototyping of 3DRTS.

5.2 VHD++ Architecture

CBD provides strong support for concurrency. It offers dynamically configurable scheduling mechanisms allowing expressing components updates and synchronization patterns in form of reusable templates. It is especially important in context of advanced virtual characters simulations, which to yield an optimal performance need to conform to certain scheduling patterns that involve mixture of synchronized sequential and concurrent updates. The Figure 4.5 presents the main elements involved for controlling the workflow execution. The runtime engine responsibilities are confined to provide set of functionalities such as late binding mechanisms through dynamic loading of components or data driven capabilities using the embedded scripting manager and XML configuration files [Price2004]. The runtime engine also provides the energy or processing time for component updates and their repartition among active threads. The scheduling mechanism is localized in the vhdScheduler. Its implementation relies on the concurrent micro-kernel pattern. The scheduler is directly connected to the inner profiler, where statistics collection can directly serve to reschedule tasks dynamically. Finally, the concurrent accesses on ISOs are ensured by the vhdPropertyManager, which use fine-grain synchronization mechanisms and its direct collaboration with the vhdServiceManager that control components execution.

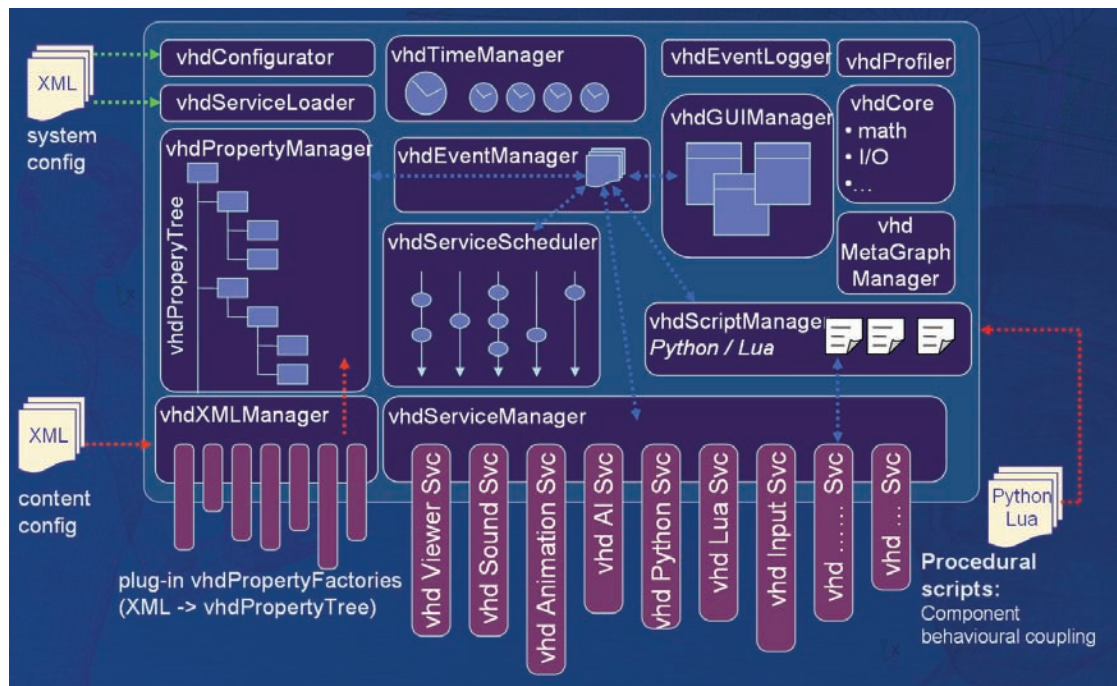


Figure 5.5: VHD++ Architecture.

5.2.1 Physical Source Lines of Code (SLOC)

To understand the scale and the complexity of our system, we use SLOCCount [Wheeler], which is a tool for counting physical source lines of code [Park1992, Jamieson2005]. The SLOC is a measure of labor and may serve to estimate the developments costs [Booch2004]. The Table 5.2 describes the statistics collected and concerns only the C++ components. We do not integrate Python and Lua scripts that differ for each simulation.

Table 5.2: SLOC statistics.

Statistics	Value
Total Physical Source Lines of Code (SLOC)	690'889
Development Effort Estimate, Person-Year (Person-Month) (Basic COCOMO model, Person-Months = $2.4 * (KSLOC * 1.05)$)	191.61 (2'229.27)
Schedule Estimate, Years (Months) (COCOMO model, Months = $2.5 * (person-months^{0.38})$)	3.95 (47.35)
Estimated Average Number of Developers (Effort/Schedule)	48.56
Total Estimated Cost to Develop (average salary = \$56,286/year, overhead = 2.40)	\$25'883'340

As we can see, the estimate development cost is beyond \$25 millions. Naturally, all the SLOC represent multiple years of software development and were developed mostly by PhD Students from both Miralab [Magnemat-Thalmann] and VRLab [Thalmann]. As a comparison, the development of the Unreal Engine 3 [Epic-Unreal] was a four years effort with a cost of more than \$40 millions [Morris2006]. Additional comparisons were made with different open-source frameworks as illustrated in the Table 5.3.

Table 5.3: SLOC comparison chart.

Name	SLOC
VHD++	690'889
OSG (without Producer and OpenThreads)	282'912
Ogre3D	182'098
SDL	120'014

5.2.2 Choosing a Topology

Bottom-up and Top-Down methodologies have their own advantages and disadvantages. In this work, we combine both approaches for defining the overall layout. The idea is to define the major elements using a Top-Down approach while being more pragmatic about existing components and performance [Hunt1999]. Thus, the design architecture will use bottom-up approach on demand. Combining both methodologies offer greater flexibility for developing the overall system. The Figure 5.6 excerpt the topology chosen. The reasons are twofold: this allows to demonstrate the system capabilities with a restraint number of components and to validate the system architecture by providing a complete vertical implementation from low-level to high-level. This serves for illustrating how the different layers can communicate together.

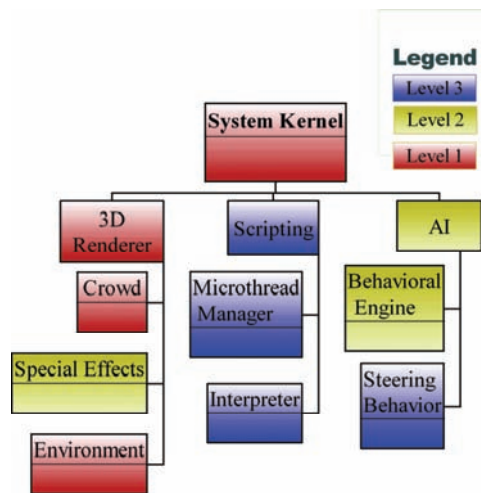


Figure 5.7: Mixed topologies using on demand components implementation.

The difference between top-down and bottom-up methodology is the way of resolving problems by emphasis on either low-level functionalities or component decomposition. Top-Down approach allow to defer low-level functionalities to clients while bottom-up ensure the system will run with existing low-level modules which may come from middleware software. Bottom-up approach allows well-factored design based on existing code base. However, this methodology also reduces the system abilities to change deeply system behaviors. But those two methodology can be combined with a mixed methodology refers as the on-demand oriented integration [McConnell2004]. The conceptual idea is to adopt both top-down and bottom-up methodology toward specific feature implementations. For instance, the creation of the 3D renderer might be built on top of existing 3D engines using a bottom-up approach. The component encapsulates the low-level graphics calls. Another example concern the development of a module handling metadata that a high-level component API could be developed before implementing the low-level parser following top-down methodology. Using this approach provide a greater development flexibility. In effect, some commercial component-based frameworks such as RenderWare [Criterion] or Karma [Kiessler] are adopting similar approach. This is aim for developers that want to use different technologies encapsulated into self-contained components, by avoiding reinventing the wheel in well-defined areas of technology. The great benefits are a higher flexibility for solving in-focus areas of technology, such as 3D rendering or physics.

5.3 Concrete Implementation

In the scope of this thesis, the resources available were limited. Therefore, the choice was made to concentrate on some elements covering the different architecture layers as a validation process. At the lower level, the goal is to extract code paths that would greatly benefit from optimizations. The intermediate layer combines and embed the different technology into self contain components easing the tasks parallelization. Finally, the high-level is dedicated to improve the system prototyping traits.

5.3.1 Intrinsic Functionalities

Since the framework target developers with various expertises in 3DRTS development, it is important that the system can provide a set of intrinsic functionalities easing the integration of components. This includes memory management primitive and concurrent data sharing mechanisms. The Table 5.4 present some features embedded into the system kernel.

Table 5.4: Intrinsic Mechanisms

Low abstraction level	Intermediate abstraction level
<i>Object referencing</i>	<i>Concurrent data sharing (mutex, monitor...)</i>
<i>Garbage collection</i>	<i>Memory management (pools, buffer, arrays...)</i>
<i>Serialization (loading / saving / transmission)</i>	<i>Asynchronous events handling</i>
<i>Active object scheduling (threads, timers, schedulers)</i>	<i>System runtime configuration (XML)</i>
<i>Time management (warp, simulation and real time)</i>	<i>Dynamic scripting (Python / Lua / CORBA)</i>

5.3.2 Low-Level Layer

To optimize the run-time performance, different low-levels libraries have been elaborated. These optimizations target to reduce the memory traffic as well exploit multi-processors architectures. This was done by re-thinking the implementation of some classical computer graphics algorithms:

- SIMD implementation of mathematics routines: most heavy processing floating points operations can be SSE vectorized.
- Early removal of hidden objects (culling algorithm) that minimize the memory traffic (Cache-Oblivious Adaptive Binary Tree)
- Objects instancing: a crowd-rendering engine where multiple virtual humans share the same geometry and textures combine with pixel shaders for improving the color variety.
- Multi-view representation: offer specialized and dedicated views on the same data using dedicated sorting queue.

5.3.3 Intermediate Layer

The intermediate layer is the centric element that combines both high and low level aspects of the system. The purpose is to ease the tasks decomposition and execution into threads through schedulers that can be configured dynamically. In our model, the decision was to compute several independent components in parallel. Ideally, for optimum performance, the component themselves should be able to run in parallel, but the time prevent us to apply deeper code parallelization. Therefore, the principal factor of parallelization is related to the subdivision of independent components, where each component remains mostly serial. This way, the system were able to integrate single-threaded component while still benefit from a multitasking execution. In addition, by offering mechanisms to specify on which thread a component will be executed, the architecture can run on different platforms (single-core, multi-core, 1-2 GPUs...) with more accuracy.

5.3.4 High-Level Layer

The high-level layer represents set of data-driven mechanisms that ease the prototyping of 3DRTS. Few developers still believe in the myth that assembly language will always produce better code. If for some intensive routines, this may be the case, it still rely that the programmers perform better assembly code than a modern compiler. The current scale and complexity of recent applications render the meaningful of implementing large code base in assembly arguable. In most scenarios, only a few 10% of the code base is using more than 90% of the overall processing power [Koch2000]. Thus, techniques like OOP or interpreted scripting languages, which were seen as generating unacceptable overheads for previous hardware, is now widely accepted due to the incremental technology improvements in both hardware and software. With these higher abstraction layers, the system potential becomes more accessible for a less-technical audience. In particular, the present architecture relies on the adoption of Python Microthreads. This extends the usability to work in a cooperative and multitasking environment. The goals minimize the technical knowledge require to work within a multitasking environment. In addition, as most of the simulations developed in the scope of this thesis manage groups of virtual humans, the need to ease the development of virtual humans' behaviors appeared. Our solution was to handle virtual humans' behaviors through their data rather than relying on the classical approach, where all the behaviors are hard-coded in C.

5.4 vhdService

The common infrastructure of the VHD++ architecture handles the simulation workflow and represents the minimum set of functionalities that are available for any application relying on this framework. The processing intensive tasks are embedded into vhdServices. Those services represent components in the CBD nomenclature. In addition, they are plug-ins loaded dynamically at run-time. The advantage is that the same executable is shared and it is possible to upgrade a deployed application by replacing an older DLL by a new iteration. Each vhdService encapsulates one technology (rendering, sound, animation...), and inherit from a common interface that is used by the system kernel to power these components. The Table 5.5 present some services that have been developed by the author and that cover different aspect of 3DRTS.

Table 5.5: Extract of vhdServices

vhdService	Function
<i>vhdViewerService</i>	High performance rendering, shutter glasses, HMD, based on OpenSceneGraph
<i>vhdSoundService</i>	Environmental 3D effects, real-time 5.1 surround sound, streaming of multiple file formats
<i>vhdAnimationService</i>	Mixing of motion generators, walk engine, keyframes, real-time motion capture
<i>vhdARService</i>	Camera image acquisition, real-time tracking, real and virtual scene blending
<i>vhdPathPlanningService</i>	Navigation graph
<i>vhdAIService</i>	Control crowds of VHs, steering behaviors, emotions
<i>vhdInputService</i>	Communication layer with input devices (gamepads...)
<i>vhdProfilerService</i>	Embed a thread-safe profiler, collect statistics
<i>vhdPythonService</i>	Embed a Python interpreter, Microthreads manager
<i>vhdLuaService</i>	Embed a Lua interpreter, efficient usage of Lua Metatables
<i>vhdOSGParticleSystem</i>	Handle Python Script to configure particle systems
<i>vhdOgreService</i>	High performance rendering, based on Ogre3D
<i>Other Services</i>	Various (shaders, physics...), application specific (GUIs, controls...)

Depending on the 3DRTS gender, developers compose applications out of vhdServices. This eases the prototyping of applications, and increases the robustness of the framework by relying on well-tested vhdServices. In addition, this offer the opportunity to equip 3DRTS with set of services and GUIs that are not deployed in the final application. The fundamental attributes that form a vhdService include encapsulating a specific simulation technology such as animation, 3D rendering, behaviors... The vhdService serve to hide the internal states and operational complexity. The Figure4.7 presents the interface of a vhdService. Apart from implementing a set of mandatory plug-in methods, each service provides a flat API describing its behaviors and a service loader use to load vhdServices at runtime based on XML configuration files. The service API is automatically exposed to scripting interfaces and since the API do not expose the inner technology, components coupling remain minimal. Finally, for services that require asynchronous computation, an optional inner thread can be launched.

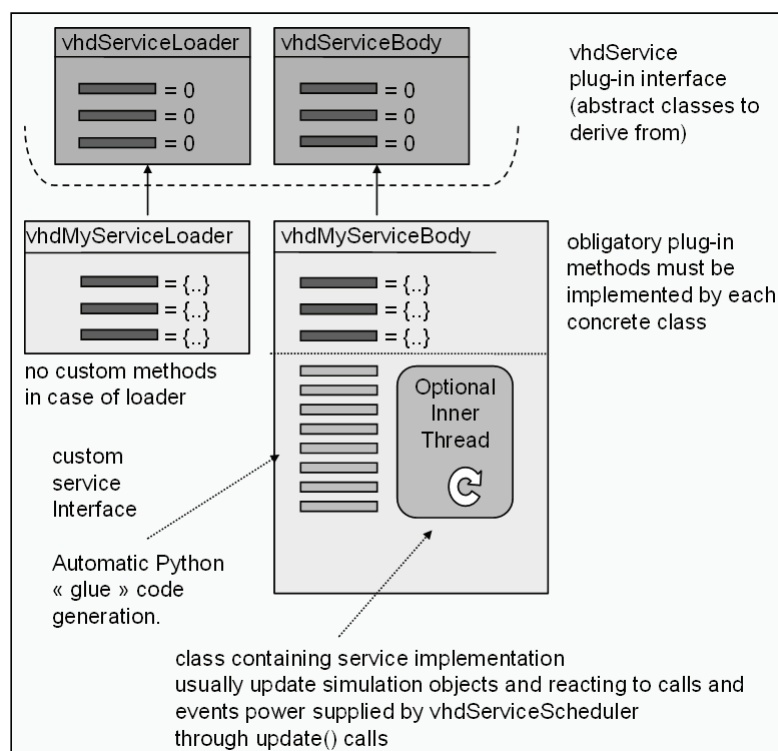


Figure 5.8: vhdService implementation.

5.5 Object Manager

The object manager is a centric element in our design. Its purpose is to handle ISOs during their lifetime as described in Figure 4.3. Its implementation rely on design patterns [Alexander1977 553]. The motivations are to provide a flexible and reusable approach, regardless of programming languages or problem domains. Each pattern intends to describe a problem that occurs repeatedly in programming environments and then describes a core solution to that problem. The idea is that the same pattern serve to resolve similar problems [Pree1994, Sane1995]. Different initiatives for categorize groups of patterns have been developed [Booch, Appleton2000, Eckel2001] [Zimmer1996]. In our case, the object manager repose on the manager pattern for handling object allocations and deletions. In addition, to clearly separate the objects from their creations, the system relies on the property and factory patterns. The factory serves to transpose a data descriptor file (XML or scripts) and send a object creation notification to the manager. Using this mechanism, the system can ensure that ISOs management is kept under control. The manager provides thread-safe assessors on ISOs and handles memory managements through memory pools and object evictions.

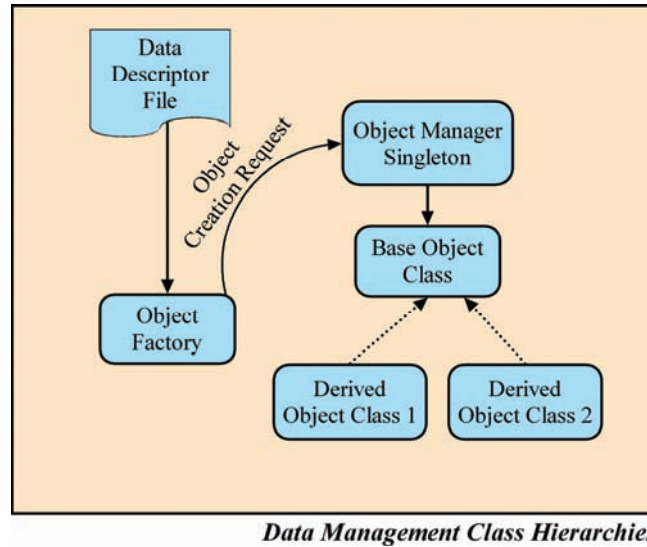


Figure 5.9: Object manager hierarchy.

3DRTS highlight situations that can benefit from patterns such as the factory, visitor, singleton, pimpl and strategy patterns [Bilas2000, Hecker2000, Dominic2001]. The reason of applying this methodology for 3DRTS developments comes in a way of reducing the complexity by describing problems with identifiable solutions. The abstractions layers and encapsulations describe with design patterns are clearly defined in the literature, which greatly help client users. For instance, [Alexandrescu2001] present the Loki library [Alexandrescu], which is a C++ library of design, containing flexible implementations of common design patterns and idioms, which may serve as a base model. The chosen terminology will directly invoke the inner behaviors without additional explanations. For our object manager we rely on the following patterns:

- **Property:** a property is an object offer set of high-level assessors on its internal data. For instance, a virtual character could be defined as a property.
- **ISO** that can be manipulated from higher-level which keep only its own data
- **Factory:** the factory pattern allows instantiating a concrete instance independently from the class definition. This gives the opportunity to clearly separate the object from their creation. Each factory are built on top of the considered property. The role of the factory is to create the property. One typical factory could receive an XML stream and create the property setting its parameters based on predefined XML tags as describe in the listing below. The advantage of applying the factory and property patterns is that we decouple simulation objects data to their envelope. This allows providing different mechanisms for controlling the same ISOs without polluting the objects implementation with extra layers of instructions.
- **Singleton:** this pattern is useful for creating manager of similar objects. In our context, singleton classes can be used for creating manager for controlling agent within the AI components or for keeping geometry in the same structure. Using this pattern ensure that a single instance of such manager exists within the system but also facilitate the ways internal object are controlled.

Listing: A XML property for creating a virtual human template and a property for one instance:

```

<vhdBBHumanProperty name="nobleM_template">
  <path> data/human/noblM00/ </path>
  <tpoFile> data/human/noblM00/noblM00.tpo </tpoFile>
  <listAction> idle 2 m_idle1 2.5 m_idle2 2.4 </listAction>
  <listAction>negative 3 neg1 1.5 neg2 4.4 neg3 2.3 </listAction>
</vhdBBHumanProperty>

<vhdBBHumanInstanceProperty name="noblM0">
  <template> template </template>
  <position> 0.0 0.0 0.0 </position>
  <orientation> 0.0 1.0 0.0 1.57 </orientation>
  <scale> 0.01 </scale>
  <strMaterialId> 0 </strMaterialId>
</vhdBBHumanInstanceProperty>

```

Adopting design patterns also help to institutionalized concepts, which reduce the errors rate by enforcing well-defined mechanisms. Surely, design patterns may not be necessary to provide a set of functionalities. Many times, alternatives can be applied for similar results, but being able to rely on them offers an important productivity boost and increases the software design maturation and code robustness.

5.5.1 Properties Manager

In our system, similarly to the work from [Filion2005], we rely on a properties manager that handle all property objects. Properties are registered at run-time, allowing activating them only when necessary. The properties manager responsibilities include the management of objects lifetime. From an implementation standpoint, the properties are encapsulated into a chained list as described in Figure 4.6. The creation of the property manager is tightly mapped on a DOM-Tree hierarchy since the architecture uses principally a DOM-Tree XML parser for creating new property instances.

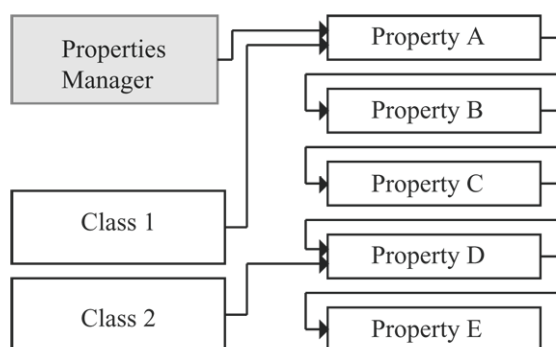


Figure 5.10: Properties manager.

5.5.2 Interactive Simulation Object Management

For accessing ISOs, simulations need to access them using unique identifier (ISOUID). As the different layer of abstractions do not have the same requirements, the internal ISOUID is stored using an integer value for optimize comparisons [Ericson2003, Kim2006], while the higher-level representations such as scripting layers will interact with ISO using strings. The usage of strings on the higher levels provides a more intuitive way for interacting with objects, allowing designers to choose meaningful name conventions. To activate an ISO, different protocols exist. This may happen whenever an ISO is entering an area or the view frustum, or automatically activated by simulation events or dynamic loading of contents. ISO activation also occurs when end-users or autonomous characters interact with objects because of input devices or scripts. Depending of the category and the expected lifetime of ISOs, they need to be ordered into groups. Some ISOs like those that the ones loaded during the system initialization are likely global. This affects objects that can be access or modified at any time. However, those objects are the exceptions not the rule and most ISOs are attached to a single node. To reduce the need to keep track of objects displacements and node transfers, a distinction between static and dynamic objects is practical. This leads to control data through a state controller layer. Creating an ISO is relatively a straightforward operation, but as some point, the system will have to free some resources including ISOs. Many techniques can be use to for ISO removal. Imagine that a virtual character throw some water on the floor and leave the room. If he comes back several minutes later, chances are that the liquid disappear. Such ISOs are good candidates to be evicted after some delay. Another situation is reflected when objects are not anymore visible and are invalidated by external events, which allow evicting these ISOs immediately. Different ISO removal protocols exist. We can remove the ISO from the simulation but keep the internal data structure into memory for later reuse. The reason behind this late removal is twofold. Some ISOs such as graphical objects should not disappear when they are still visible and better cache management of common ISOs can be achieved by keeping the internal data structure intact. In addition, some ISOs may be critical for the system and should not be remove at all. The way ISOs are managed is directly connected to the memory management layers including memory pools. They reduce the memory fragmentation that affects the constant creation and destruction of ISOs, which help to minimize problems such as paging to disk or seek time [Denman2003]. The techniques involve for managing ISOs include:

- *Fluff removal*: set of properties that allow providing information on the probability of reusing the same ISO in a short period. For outside environments, it is generally useful to keep memory graphical elements such as skybox in the cache, even when the simulation is temporary entering in a building. This is done by assigning those ISOs with high probability of reuse. Some less critical objects, such as eye-candy graphical elements can be ignored or removed when resources are limited. This increase the system flexibility by running the same simulation on different hardware, where non-interactive objects are rendered when processing time is available.
- *Tiny Objects*: represent extra elements that do not interfere with the interactive simulations directly. These ISOs are designed to exploit hardware that is more powerful without affecting the simulation workflow. Such tiny objects are created on the fly and using buffered memory data structures. Such objects are mainly graphical ISOs but also include complex entities (butterfly, extra lighting effects...). The system dispose of these tiny objects on its own and decide whenever to create or remove them. Here, the use of buffered data structures is vital, as up to 80% percent of visible ISOs can be tiny objects, which may rapidly fragment the memory and reduce system performance.
- *Expiration*: represents ISOs that stay in the memory for a specified time since the last time they were accessed. Typical examples, are bullets, fruits... that can be deleted after some delay, freeing valuable resources. An alternative consist to store those clonable objects into memory pools that kept a certain amount of instances instead of using a timing policy. In our design, all ISOs inherit from this property but the expiration can be set to infinity, avoiding unrequited removal of ISOs.

As ISOs can interact with each other, some objects may be remove from the system but still connected to another one. This may involve system crash if those issues are not handled accordingly. To prevent that the logic become inconsistent, the usage of shared pointers on data structures enforce that no objects are deleted inappropriately [Meyers1997]. The system is responsible to propagate the information on objects removal.

5.6 Data-Driven Architecture

The development of 3DRTS resemble to a never-ending story. There are always new technologies and improvements to be made and without the pressing of milestones and deadlines, no software would become available. Sometime, the development process and the system abilities to respond to modifications may cause its failure. The developments of interactive simulations do not differ greatly from other software developments from a design perspective. Because of the high competition and the continuous technological race, much systems architecture for 3DRTS is not applying some of the successful methods developed in other fields of software engineering. The principal drawback is that most of the development time is spend on building the core technology rather than focusing on the simulations content and interactions. Finding the proper interactions model is an iterative and difficult process. It requires adapting simulations parameters to specific conditions. For instance, if we want to simulate a virtual character guiding end-users in an environment, we may try to choose some average walking speed based on real charts. However, navigating within a 3D environment is not completely reflecting the real world. Many parameters like the input devices may alter the relation to speed and timing. In such a scenario, simulation designers have to modify the systems for improving the interaction model. In a classic system, those values may be hard coded as C++ code. It forces programmers to edit the source code and recompile the application. If every time, designers need to modify a parameter, this whole procedure is applied, the development and simulation tuning will quickly become unmanageable [Shumaker2004]. To overcome these limitations, many systems, which claim to be data-driven, are creating configuration files for moving those parameters outside of the source code. This avoids a full recompilation every time parameters changes. Designers are able to modify the values without the assistance of programmers. Even so, such systems can be parameterized; they remain highly dependent of the information exposed. If some parameters are missing, they may not be change directly. Data-driven architectures as the one presented here goes beyond these limitations by clearly separating both the logic and data layers [Riley2003b, Malafeew2005].

The ability to implement the simulations functionality outside the scope of the logic components is one of the important benefits of Data-Driven methodology. The code maintainability traits are clearly improved as it avoid that scenario specific code paths affect the code base. Moreover, the scope and size of such systems have some real shortcomings. Even so the dependency graph between the modules and library remain minimal, it is common to observe that a full system recompilation may take 5 to 20 minutes. Certainly, incremental linking

and others compiler speed-up optimizations can reduce this time, it still results in many frustrations to work with systems that bind both the data with the code base. This reduces the ability to fine-tune the simulation because of the compilation overheads. Thus, being able to control from a higher level, the simulation data and events improve the platform potential. In our architecture, the simulation data can be manipulated through different layers including interpreted scripting languages. One difficulty related to the use of interpreted scripting languages is that a wrong usage of scripts may kill the system performance. The difficulty relies on offering adequate mechanisms that facilitate to spread tasks without strong technical knowledge. Thus, the system has to be capable of handling most of the technical issues like memory management and multitasking operations. This is primordial as simulation designers may not have the abilities to handle low-level problematic.

5.6.1 Logic Layer

The logic layer represents what many systems refer to system engine. This is where the underlying system controls the different aspects of the simulation. Those subsystems are services packages in CBD. They encapsulate the low-level and specific code implementation and define components such 3D renderer, sound system, resources management, or math library.

5.6.2 Data Layer

The data layer is the simulation code that contains all the higher level of interactions that can change without modifications to any run-time library or executable. The candidates' modules include the behavioral rules for controlling agents, scripts for dispatching system events like manipulating special effects, such as clouds or particle effects but also for providing access to all the Logic Layer API. The simulation is being fully controlled by the Data Layer. The Logic Layer does not change for every scenario. Its purpose is to process data in efficient ways without knowing anything about their meanings. For instance, the Logic Layer would be able to process Finite State Machine transition without information on the states themselves. A good data-driven system should make the distinction between the two layers as strong as possible. Data Layer relies on top of the Logic Layer, which indicate that it can know the underline layer but not vice versa. The Data layer is the only element in the architecture, which is responsible for retrieving and controlling data. The Logic layer may receive those data and process them but it will not directly interact with the data.

5.6.3 Data Manipulation

Data used within the system has to be seen as token and elements within the simulation. For instance, such elements can be a character, a sound, or simulation events. The system management is a self-contained module that operates directly on the data. It provides the energy to the system by implementing simulation behaviors. The system should not be bind to particular data but rather be built around functional system aspects, as many elements will interact with each other. Thus, it is better to see the system from higher perspectives for twofold. The system will be more easily maintainable and the higher level of abstractions free designers of low-level technical issues.

5.6.4 Interactive Simulation Objects (ISO) Structures

As the Logic Layer does not have the knowledge on elements to be created, there is a need to implement a methodology for instancing ISOs. One pattern that fulfills this requirement is the Factory pattern. It consists of a function that can create low-level objects on demands. Base on data, the factory pattern generate a new property or simulation object. The system kernel is responsible to send events to each interested service, informing them that a new object was created, modified, or deleted. Then the services propagate this information to the low-level components such as the 3D engine. The factory pattern insulates concrete details from the systems to spawn entities.

5.6.4.1 Object-Centric vs. Component-Centric

These two methodologies approach creating ISOs using either static or dynamic hierarchies. Static hierarchies are hard coded in C++, which provide efficient run-time performance but lack the flexibility for future design decisions. The Figure 5.11 describes hierarchy where coders need to provide all the different concrete implementations in code, using inheritance. Inheritance prevents duplication of data, but may have a

performance impact on cache-size limited platforms. Some programming optimizations tricks exist but reduce options for changing archetypes.

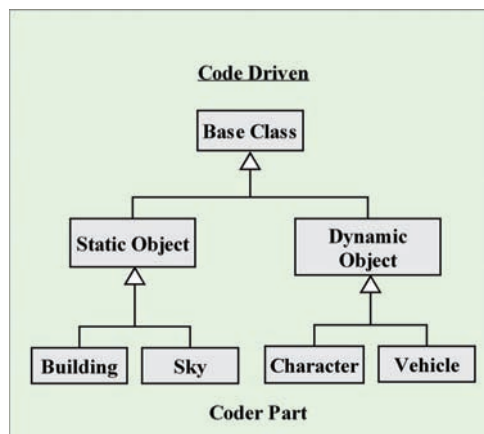


Figure 5.11: Static classes' hierarchy.

Generally, it is also better to avoid deep inheritance hierarchies. The shortcoming of too heavily inheritance-based class structures is reflected by the lack of flexibility in responding to changing requirements. One typical problematic is refers as the “*Diamond of Death*” that appears when a class has the same (direct or indirect) base class appearing more than once as an ancestor. The Figure 5.12 depicts such occurrence. As a result, inheritance from interfaces is good while inheritance from concrete classes should be avoided.

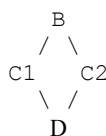


Figure 5.13: Example of a “*Diamond of Death*” shape.

In contrast, dynamic hierarchies describe the objects descriptions later as shown in Figure 5.14. The idea is to keep the basic property in code and to let designers defining object behaviors using configuration files or scripts that create ISOs on the fly. This mechanism facilitates late design decisions and gives more control on the ISOs lifetime, by simplifying adding and removing ISOs. This data-centered style can be seen as a way of storing data where clients' objects connect and operate directly on the data [Stoy2006]. This offer a way to separate the object behaviors from the object storages, making theses two elements independent of each others as analyzed by [Bass2003]. Dynamic class hierarchies are less coupled than static hierarchies but this come at a performance cost [Duran2003, Church2004]. However, dynamic hierarchies are more suitable with C++, which do not support well deep inheritance levels. Thus, systems architectures may try to combine and balance both approaches when designing simulations objects.

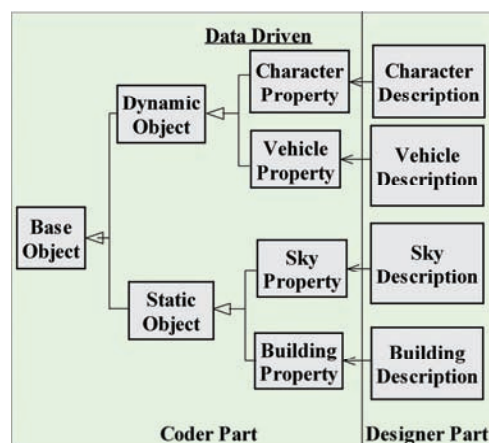


Figure 5.14: Dynamic classes' hierarchy.

5.6.4.2 Basic Object

Every native object from our system inherits from a basic object call *vhdVoid*. This basic class encapsulates the different information that is needed by the system for managing concurrent access and memory allocation of data [Llopis2004]. The listing above exert the principal elements stored

Listing: vhdVoid

```
vhdVoid
{
protected:
    Mutex        _refMutex; //mutex use for lock acquisition
    uint32        _uiRefCount; //reference counter;
    uint32        _uiLastTick; //last access (use as timeout)
    int32         _uExpiration; //object timeout
    eReuse        _eReuseProbabilty; //use to avoid (delay) unloading
    ePriority      _ePriority; //can we throw it if needed?
    eType         _eType; //static, dynamic, global ,...
    eCategory     _eCategory; //graphic obj, sound, ...
    bool          _bLoaded; //for streaming
    bool          _bTiny; //for small object allocator
};
```

5.6.4.3 vhdProperty

vhdProperties are an extension of the vhdVoid interface. They encapsulate generic data (configuration data, 3D models, animations, sounds...) and ensure the validity of the internal states through dedicated thread-safe assessors. Their role consist to describe ISOs and to ease the integration of new data types by client developers. vhdProperties form a treelike structure to express generic dependencies. For instance, a virtual human property may have sound sub-property that will be automatically updated whenever the virtual humans move. A vhdProperty represent shareable data (for concurrent access) and result from the combination of inheritance from stable interface and data-driven specialization provided by a vhdPropertyFactory. The following listing extracts a vhdProperty.

Listing: vhdProperty

```
class VHD_SOUND_DLL_EXPORT vhdSoundServiceConfigProperty : public vhdProperty
{
private:
    //fields
    vhdSoundEngine *_pSoundEngine;
    //...
public:
    //getters
    const std::string & getSpeakerConfig();
    const bool & getUse3DSound();
    const vhtInt getSoundThreadPriority();
    const float & getDistanceFactor();
    const float & getListenerRolloff();
    //...
    //setters
    void setSpeakerConfig( const std::string & );
    void setUse3DSound(const bool & );
    void setSoundThreadPriority(const vhtInt );
    void setDistanceFactor(const float & );
    void setListenerRolloff(const float & );
    void setVolume(const long & );
    //...
};
//_Impl
void vhdSoundServiceConfigProperty::setSoundThreadPriority( const vhtInt iPrio)
{
    vhdSYNCHRONIZED(this); //thread-safe access
    _pSoundEngine->setThreadPriority( iPrio );
}
```

5.6.5 Decoupling Information

The simulation tokens can represent complex objects like virtual humans. Characters are made of several elements. They must be rendered and animated and should feature realistic behaviors. From a higher-level representation, characters are seen as single entities. However, it is primordial the information is spread into different low-level modules. The reason of this clear separation is that large portions of functionalities are likely to be found on several groups of tokens. In addition, some elements like the 3D renderer require the data to be sorted in very strict order for optimizing the graphic rendering pipeline. On a lower to middle level perspectives, the system must be separated into smaller components, allowing to reuse as much as possible functionalities while the higher level see them as single entities. The limit of this separation is reached when coupling elements is required. Ideally, smaller elements should be self-contained so they can be tested separately.

5.6.6 Communication Interfaces

The Logic Layer has to provide clean interfaces that follow advanced software engineering practices targeting code reuse [Stroustrup-Reuse1996]. The first solution would be to use C++ inheritance for sharing functionalities and interfaces. If for some components inheritance may work perfectly, it does not scale well [Meyers1995]. Having too many levels of inheritance, produce complex code and it may not perform well on cache size limited platforms. In addition, the code robustness is limited as a modification on a top-level interface may have deep effects in the code base. Finally, it may be difficult to add more subclasses as they may regroup much functionality from diverse branches. In our architecture, we rely on an alternative organization called the component-based system. It uses containment pattern. Instead of encapsulating the system hierarchy into a multilayer representation, every layer can communicate with the lower level while the higher level does not need to know anything from the bottom layers. This approach offer better maintainability and extendibility traits as it prevent bidirectional communications between the different layers resulting on more streamlined developments process. CBD provide systems that encapsulate and separate components into simple layers. It frees the developers from knowing the inner logic of each component, while allowing the object composition to change at runtime. For instance, a virtual human can derive from an animation interface that the animation component can handle and from a geometry interface used by the 3D renderer to display the character. This way, the definition of a virtual human is hidden for both components. They only need to handle part of the data. This indirection offers the opportunity to come with new property that is automatically manipulated by existing components while extending the property with specific needs. The following listing illustrated this approach.

Listing: Handling properties

```
// 3D graphics renderer component (extract)
vhdRendererService
{
    // the kernel notify the service whenever a property is altered
    vhtBool _handleAddPropertyScanImplem( vhdPropertyRef property)
    {
        vhdSYNCHRONIZED(this);
        if ( vhdREF_IS_OF_CLASS( vhdGeomertryProperty, property))
        {
            vhdGeomertryPropertyRef geomProp =
                vhdREF_DYNAMIC_CAST(vhdGeomertryPropertyRef, property);
            // create the internal graphical object
            shared_ptr<Geometry> pGeom = new Geometry(geomProp->getVB(), ...);
        }
        return true;
    }
    vhtBool _handleRemovePropertyScanImplem( vhdPropertyRef property)
    {
        vhdSYNCHRONIZED(this);
        if ( vhdREF_IS_OF_CLASS( vhdGeomertryProperty, property))
        {
            vhdGeomertryPropertyRef geomProp =
                vhdREF_DYNAMIC_CAST(vhdGeomertryPropertyRef, property);
            //remove the internal object ...
        }
        return true;
    }
}
```

```
};

// customized property for a virtual human
// the animation, physic and 3D renderer will interact with this property,
// using their own specific interfaces. The complete character property will be
// handle by the AI Component
// interfaces while the complete vhdCharacterProperty will
class vhdCharacterProperty : public
    vhdIMovable, vhdGeometryProperty, vhdAnimationProperty, vhdPhysicProperty
{
    //extra data (behaviors, steering attributes...)
};
```

5.6.7 Data and State Controller

Data-driven design for controlling simulation objects gives the ability to separate both the simulation objects controller from the system as shown in Figure 5.15. The underlined system does not need to worry about the ISOs themselves but need only to know how to compute them. This gives the ability for each simulation to choose how they will handle the ISOs management. For instance, for a graphical object, the underlined system need to organized the data for rendering the objects while the simulation states controllers will apply modifications on the objects themselves. Thus, whatever internal representation is used by the system, the simulation layer can still provide another view on the same dataset. For instance, even so, the system may classify 3D objects using an Adaptive Binary Tree (ABT-tree) representation for the scene management, the simulation layer still have the opportunity to represent the same scene using a scene graph representation.

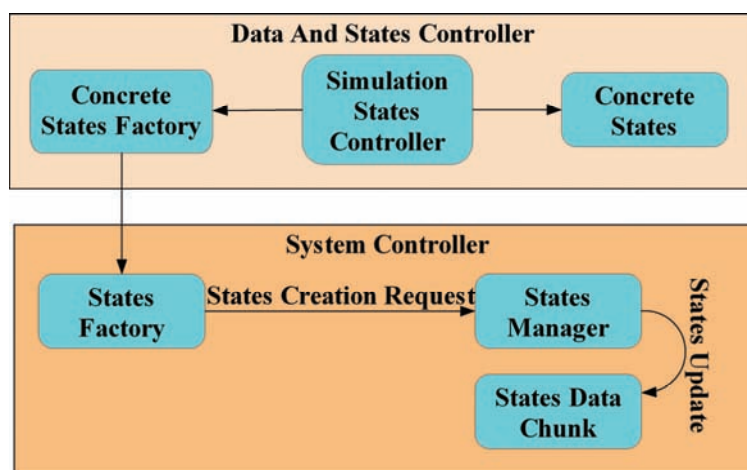


Figure 5.15: The concrete data and states controller is separate from the underlined system controller, which only keep a memory representation of the considered simulation object.

5.6.8 Late Binding Mechanisms

CBD combine reusable elements. In general, such systems offer more components than needed by a single application. To select the appropriate components, different approaches exist. One solution consists to compile a different executable for each application, which will be statically linked with some components. This obliges to provide different runtime executable for each configuration. Using this methodology, developers often rely on providing a unique executable which contains all possible components avoiding creating distinct applications. This increase both development times (compilation, dependencies, etc.) but also impact on the system performance with an increase memory footprint. A better approach is to promote components that are compatible with late binding mechanism. This refers to pluggable components. A pluggable component derives from a common interface that is known by the system micro-kernel. It allows loading and unloading components dynamically. This gives the opportunity to create a unique executable that contains only the micro-kernel and to let the system load dynamically components on demand. In our system, we rely on XML configuration files that specify which components to load. This approach promotes the capabilities to run the same application with a limited number of components, which greatly improve debugging sessions by isolating components one by one.

5.6.8.1 Dynamic Component Management

Every component in the system forms a dynamic library. The management of DLL is dependent of the platform. Every platform that support dynamic libraries provide functionalities for loading and obtaining address to objects. To hide the platform specific calls as well as to keep a trace of DLL management, a small libraries handling DLL management have been developed. It consists of a dynamic libraries manager that encapsulates platform specific calls as describe in the listing below:

Listing: Dynamic library manager:

```
#ifndef WIN32
#define DYNLIB_HANDLE hInstance
#define DYNLIB_LOAD( a ) LoadLibrary( a )
#define DYNLIB_GETSYM( a, b ) GetProcAddress( a, b )
#define DYNLIB_UNLOAD( a ) !FreeLibrary( a )
struct HINSTANCE__;
typedef struct HINSTANCE__* hInstance;
#else //assuming linux
#define DYNLIB_HANDLE void*
#define DYNLIB_LOAD( a ) dlopen( a, RTLD_LAZY )
#define DYNLIB_GETSYM( a, b ) dlsym( a, b )
#define DYNLIB_UNLOAD( a ) dlclose( a )
#endif

void vhdDynLib::load()
{
    m_hInst = (DYNLIB_HANDLE)DYNLIB_LOAD( _strName.c_str() );
    if( !m_hInst )
        std::cerr << "Can't load DLL: " << _strName << std::endl;
}

void vhdDynLib::unload()
{
    if( DYNLIB_UNLOAD( m_hInst ) )
        std::cerr << "Can't unload DLL:" << _strName << std::endl;
}

void vhdDynLib::getSymbol( const std::string& strName ) const
{
    return (void *) DYNLIB_GETSYM( m_hInst, strName.c_str() );
}
```

5.6.8.2 XML Files

An XML file is used during the system initialization, which allow specifying which module needs to be loaded at runtime using dynamic loading of components mechanisms. The listing below describes an example of XML tags for loading components and GUIs using late binding mechanisms.

Listing: Code snippet from XML configuration file specifying which module need to be loaded at run-time

```
<plugins>
  <service className = "vhdPythonService" name = "pythonService" />
  <service className = "vhdHAGENTService" name = "hagentService" />
  <service className = "vhdOSGService" name = "osgService" />

  <vhdGUIConfigProperty name = "vhdPropertyManagerGui"/>
  <vhdGUIConfigProperty name = "vhdObjectControlGui"/>
</plugins>
```

At run-time, the kernel is responsible to parse XML configuration files indicating the *vhdServices* to be loaded dynamically. This way, a unique executable is handling all the different simulations. This facilitates the maintenance and expansion of additional plug-ins for older simulations. This may lead to replace older simulations with newest components implementations.

5.6.9 Scripting

One drawback with CBD and the large number of components associated is that each of them comes with very different behaviors. Some components like AI (Artificial Intelligence) may deal with large number of data and parameters, which require many tweaking for proper usage. For instance for every scenario, characters will have different capabilities that need to be reflected on the screen. If we want to customize the simulation, we need to treat much functionality as data rather than code. By exposing the system capabilities within a scripting languages offer an immense amount of flexibility that allow faster prototyping of applications and also give more power to simulation designers. The main idea is to keep time critical functionalities as optimized C/C++ code but to control and interact with the environment within a higher-level language [White2001].

5.6.10 Flow of Data

One benefit of Data-Driven design is that they simplify the capabilities to manipulate data on the fly. It act as a pipe and filter, and offer better capabilities of reuse and are more easily to maintain [Calvert1996]. It becomes possible to load and unload large portion of the environment dynamically, by simply uploading data on demand [Bilas2002]. This reduces development time by avoiding restarting the whole application when some small modifications on the data level are required. However, one problem with data flow is that high-level ISOs may operate on different low-level functionalities. For instance, a character will be composed of an AI entity and a 3D mesh representation. The data pipe that bind those simulations objects need to propagate the information to different components, interconnecting them to different logical layers. This may induce a significant overhead parsing and dispatching tasks [Acton2005]. However, data flow combine components using data-driven mechanisms, which control high-level simulations objects relying on different low-level components.

5.7 Data Streaming

For loading simulations assets, many approaches exist from binary files to ASCII files formats. In our system, we choose to adopt intermediate metadata files that will serve as creating platform specific data as an off-line process. If the direct loading of content from metadata files is used during the development, we encode the content and simulations assets into an efficient system memory layout using custom tools such as [osgConv] for simulations deployment. This first approach using metadata at run-time offers a more streamlined methodology as it does not rely on particular hardware [Llopis2006]. However, this method has some serious drawbacks that increase loading time. As the inboard memory features within current hardware has significantly increased, the storage devices intrinsic performance did not evolve as fast. This inducts that filling the memory with data is longer [MSDN_LT2002]. To decrease the loading time, the system should avoid unnecessary data manipulations and being able to read significant amount of data store locally in space at a specific time [Bilas-ISL2001, Koenig2006]. This obliges the system to provide an abstraction layer on files systems operations. Every platform works with different files paths systems and different disks. For instance, we can take benefits of intrinsic file system functions for asynchronous files operations [Lee2006]. The problematic of working with different storage disk is principally affecting streaming algorithms, where seek and bandwidth performance may have a critical impact on some platforms where data may not be ready. Using specific files for each platform reduce loading time. It facilitates to profile I/O performance as developers can distinguish if the problem is directly related to I/O operations rather than on parsing and converting data. Moreover, dedicated a background thread for loading and unloading continuously data allow the system to react in a shorter time. The Figure 5.16 describes the platform specific data loading retained in our design.

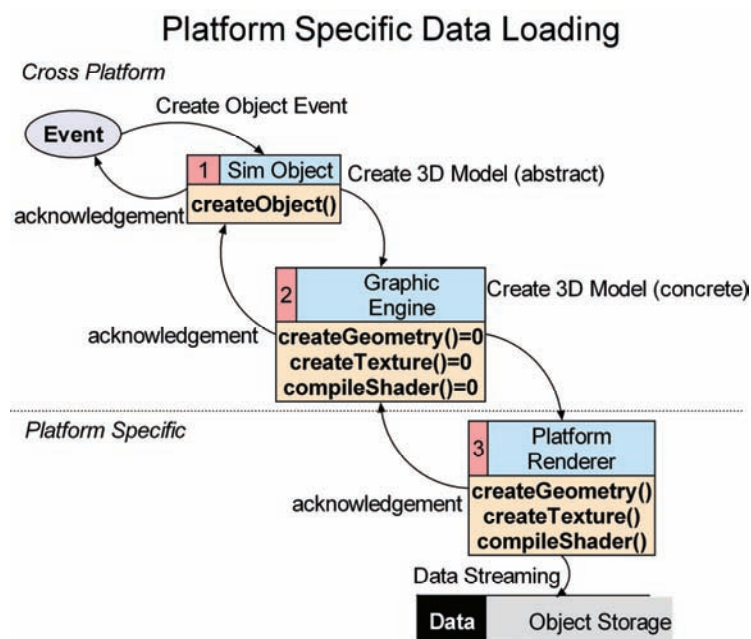


Figure 5.16: Data loading from storage devices.

The organization of the binary files mirrors the internal code representation but may also be store within archive files. In fact, some OS and antivirus software may increase deeply the time for obtaining a handle to a file. By using a set of archives opened during the initialization phases, the system can perform I/O operations more accurately. Using specific set of data by platform provide assets in a format that fit the hardware performance. For instance, some recent hardware is supporting normal map compressed textures while older cannot handle such files. Using dedicated files avoid converting those files on the fly.

5.8 Serialization

Software architectures describe the fundamental organization that interfaces the components. Their relationship and the principles guiding their internal design and evolutions are closely related to their internal functionalities. For 3DRTS as for many others applications, there is a need to save and load the simulation content. Because of the extreme connectivity of components and important number of dynamic components, it is not always doable to implement functionalities to save and load the system at any point. System may save only on predefined checkpoints. Nevertheless, when writing the code, some mandatory code helping to provide such functionalities includes serialization and objects persistence. Generally, data structures use by 3DRTS contains many “hot”-structures that encapsulate pointers [Ericson2005]. The serialization implementation is easily prone to errors and very sensitive to the context. Hopefully, OOP methodology is reducing this problematic by defining data into objects. They are instances of classes, which describe the attributes and behaviors of these objects. This facilitates the serialization process by encoring the object memory representation into another representation. Serialization can be used for saving and loading data as well as for persistent objects. [Duran2003] pointed that saving and loading have to be separated from exporting and importing data as they have different needs. He also describes that saving and loading procedure should be handling differently. Saving and loading should involve minimal data. It should compress simulations states as much as possible and almost certainly keep it in binary form. Exporting and importing, on the other hand, needs to replicate an object in all its complexity. It conveys maximum states information, needs to be versionable, and may benefit from being text-based, so data remain readable. In our architecture, the different properties defining simulations objects that can be manipulated with higher-level languages such as virtual characters or environments provide serialization functionalities. The concrete implementation relies on the Boost Serialization Library [Boost]. This library represents a modern open source and cross-platforms initiative. As other Boost components, the Boost Serialization Library rely heavily on metaprogramming [Koenig2000, Abrahams2004, Karlsson2005] taking advantage of C++ templates for generating the dynamic binding using clever scheme using the C++ typeid facility. It provide set of functionalities to write different serializers but come with default mechanisms for writing into binary, text or XML files formats. The listing below illustrates a class serialization using the boost::serialization library:

Listing: Serialization

```

class VHDEL_DLL_EXPORT iCharacter: public iBaseEntity
{
    friend class boost::serialization::access;
    template<class Archive>
    void serialize(Archive & ar, const unsigned int version)
    {
        //serialization of upper class.
        ar & boost::serialization::base_object<iBaseEntity>(*this);
        ar & *_pPath;
        ar & _strCurrentPlaceName;
        // ...
        ar & _bCondition;
    }
};

```

5.9 Scene Representation

For managing 3D environments, different techniques exist. One methodology that offers good high-level scenes representation is the scene graph [Blythe2000, Eberly2000]. However, scene graph rendering APIs come with several limitations. The generic approach of 3D scenes management with the typical “all-in-one” rendering-culling-animation-display black-box approach is not suited for scene graph independent architectures. Instead, the idea is to control 3D environments through an independent meta-scenegraph [Bar-Zeev2003]. With it, we can promote optimize and dedicate rendering, culling, declarative scenes specifications, animations, etc. graphs. As [Arnaud1999] say: “*The scenegraph is a victim of its own success: it works so well as a graphics abstraction that it has been pressed into further service as an application abstraction, which it is not*”. “*We can overcome the limitations of well-know scenegraph concepts, if we distinguish between structures and contents of a scene specifications. A single generalized scenegraph can map its contents to different rendering systems...*” [Döllner2000]. If we analyze the different scene managements based on scenegraphs, some key elements can be excerpt: they are all trees (directed acyclic graphs) compose of heterogeneous nodes and leaves. Every node has a set of fields that contains the node's attributes. The revolutionary notion of scenegraph for 3D graphics applications was first introduced by SGI in 1992 with the paper from [Strauss1992]. Since then there was no significant break-through reported and scene graphs have been effectively used as macros for lower level 3D APIs calls [Arnaud1999] (lights, textures etc.) as well as hierarchical structure elements (groups, transforms etc.). Experiments highlighted the fact that scene graphs (Performer, Optimizer, Cosmo3D, VRML97, OpenInventor etc.), facilitate rapid data-driven programming of scenes. However, if they are over-used in an all-in-one approach so that their hierarchy and execution semantics are overloaded (scope-of-effect, order of evaluation, routes, fields, etc.) then they are featuring multiple clashes [Arnaud1999] due to architectural constraints when new algorithms (e.g. mirror, fragment shader rendering) are required. Furthermore, with the latest advent of high-power consumer graphics boards and the explosion of 3DRTS, the traditional scenegraph notion has prove to pose multiple performance bottlenecks for larges scenes [Jones1999] and thus are not widely use in modern 3DRTS engines. However, as our architecture do not depend of the presence of any component including the 3D renderer, we can not opt for the classical approach where the 3D renderer is responsible to organize the scene. Thus, we need to come with a solution that is agnostic to specific requirements.

5.9.1 Related Work

From the previous scenegraph APIs, two emerging categories are striving to evolve the traditional Inventor model [Strauss1992].

1. *OpenSG-Jupiter scene graphs*: Keep one scenegraph and employ intelligent traversal through dedicated agents that decide which nodes are active or not, so specialized iterators can traverse them [Reiners2002].
 - a. *Drawbacks*: although this works for nonstate-dependant traversal (e.g. statistics), it is not appropriate for e.g. RenderAgents as they need the scenegraph topology, thus few alterations-optimizations (overall states sorting) can be easily performed. It does not handle multiple renderers.
 - b. *Advantages*: the while scheme is extendable and allows to easily add new traversals and nodes without altering the library sources.
2. *Generalize Scenegraph-Virtual Rendering System-Open SceneGraph*: they create one generalized scenegraph and then allow different rendering engines to be used on the same content (OpenGL, Renderman, etc.)
 - a. *Drawbacks*: cannot extend easily with new nodes without changing the source code, does not handle events, animations, etc.
 - b. *Advantages*: can employ multiple renderers, OpenSceneGraph is the first to feature vertex shader programs, as an added state attribute (appearance node).

From the two methods above, it seems there is no single scene graph where it can be extended easily without changing its original source code. Hierarchical transformation graphs (current scenegraph) and spatial graphs (BSP-tree, dPVS) which are the most optimized solution for animations, and rendering-culling respectively represent the two major scenes management categories. For combining both aspects, we develop the idea of a new multiview separating meta-graph for maximum performance. Some drawbacks include the additional synchronization layer among graphs but extend the ability to provide high-level scenes representations and efficient rendering organizations simultaneously. From this perspective, we can see that all-in-one scenes management solutions like [BMBF, Osfield, SGI, Strauss1992, Cosmo3D1998, Bartz2001, Berthel2003] cannot meet all the above requirements for advanced real-time rendering. The current trends within commercial game engines such as [Epic-Unreal, idSoftware] is to separate the scene graph into meta-graph. The innovation lied in the tight integration and cooperation providing a multiview scene representation, which allows decoupling rendering from the simulation itself. The primary objective behind this approach is performance tuning and flexibility, so characters animations and static scenes rendering could be accesses through different paths, allowing both optimizations (state sorting, texture management...) [Chenney2002] and high-level assessors (access by id, name,...).

5.9.2 Meta Graph Manager

The innovative solution that combines both high performance of spatial graph and high-level representations of scenegraph approach is the meta-graph approach. This approach contains various scenes representation as depict in Figure 5.17. This allow the creation of specialized view-calculations (rendering, culling, animation etc), on datasets among specific interests. The motivations are directly connected to our architecture. In our model, the common infrastructure is not bound to any component, including the 3D renderer. Therefore, it is impossible to adopt the classical approach, where the 3D renderer is responsible to organize the scene elements. Here, the meta-graph serves to shared data as well as offering the opportunity to optimize the objects classification for more than one component.

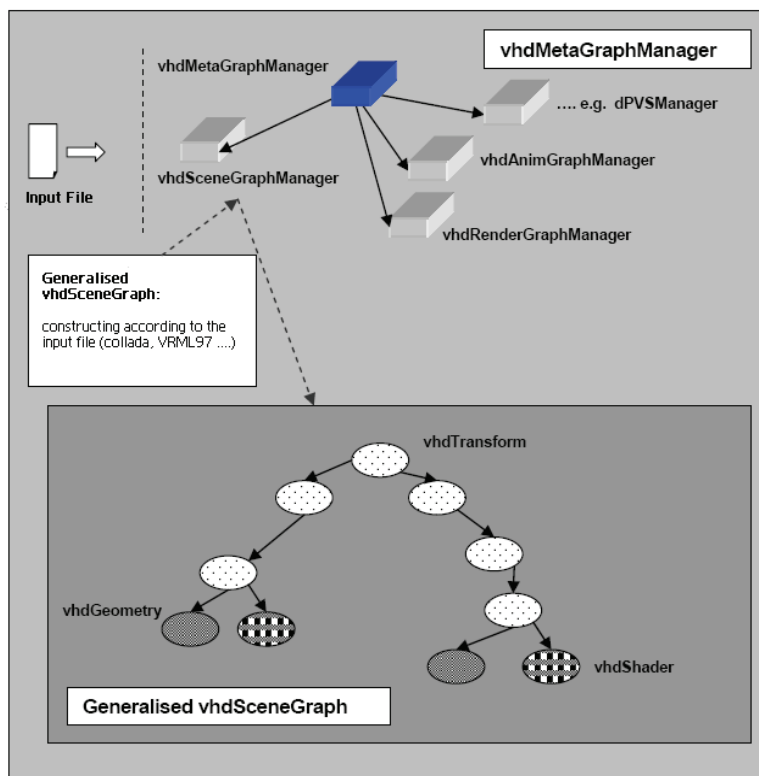


Figure 5.18: The Meta Graph Manager.

The meta graph manager relies on the Strategy and Composite design patterns [Gamma1995]. Here, the `vhdMetaGraphManager` is responsible of grouping separate meta graphs, as these are construct from the common `vhdMetaSceneGraph`. The lower property node represents shared objects use in multiple contexts. It can act as an independent object in each context or subgraphs (see Figure 5.19 and 5.20). These two samples illustrate in detail the generalization of a `vhdMetaSceneGraph` where each graph share the same data, encapsulated into a `vhdNodeProperty`. For instance, the Figure 5.21 presents the animation meta-graph and the Figure 5.22 presents render meta-graph. The difference between those two meta-graphs is the way data are sorted. The animation meta-graph sorts its children according to their transformation matrix field (translation hierarchy) whereas the render graph sorts them according to graphic states (that encompasses Shader Programs, Textures, Materials etc.) The animation graph picks from the generalized `vhdSceneGraph` the node that forms a translational hierarchy (e.g. `vhdTransforms`). It is responsible of updating the transformation matrix of each node in the pool (global transform).

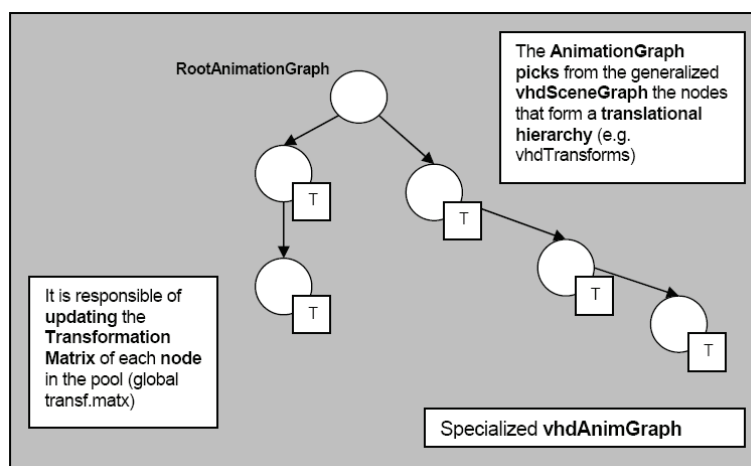


Figure 5.23: Concept of a `vhdAnimGraph` where each node is shared (`vhdNodeProperty`).

The render graph picks from the generalized `vhdSceneGraph` the Geometry nodes and sorts them according to their render states (e.g. shader programs, textures, materials, lights...). In our implementation (see Figure 5.24), we rely on a cache-oblivious implementation of an ABT tree for deciding which node is visible. It is responsible of rendering the whole scene according to Visitor-Iterator pattern traversals.

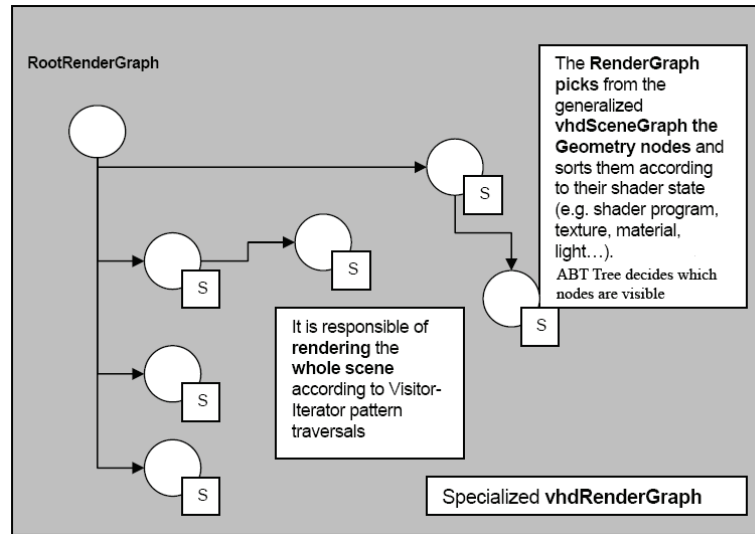


Figure 5.25: Concept of a `vhdRenderGraph`, optimized for fast rendering.

5.9.3 MultiView Scene Representation

3DRTS such as crowd simulations require addressing many problems from different perspectives. Typical simulations may feature more than a few hundreds distinct virtual characters. The number of assets to be created and controlled interactively within the simulation is becoming a predominant factor. Naturally, the real-time performance of such a virtual environment depends on the appropriate usage of a 3D API such as [OpenGL]. These APIs come with design particularities that oblige the developers to organize their rendering pipeline in a strict order. Unfortunately, this organization is not intended to be easily controllable for high-level simulations designers. Often, developers have to make some compromise between real-time performance and the flexibility offered by their system. To overcome this barrier, an alternative approach for scenes management can be applied by extending the way the data can be fetched. The idea is to provide different views based on module specific requirements.

By providing different view representations of the same-shared data, we can optimize both the access time and the different components computations. For instance, the GPU rendering pipeline and the semantic representation have distinct requirements. The 3D rendering module needs to classify 3D meshes among their rendering cost penalty (OpenGL states sorting, shaders parameters bindings, etc) [Dietrich2003, Cebenoyan2004, Wolka2004], while the semantic view may access them by their unique ID. Our approach overcomes the limitations of classifying objects in a unique list or queue. To implement this multiview representation, we are relying on the multi-index container from the [Boost] library. This container maintains multiple indices with different sorting and access semantics. It goes beyond the single view representation the STL map or set may offer. The original idea of this concept came from the indexing theory as used in relational databases. Thus, we can simultaneously optimize data access for spatial, states or semantic considerations, by fitting precisely their local requirements. Figure 5.26 depicts such a container with three view indices on the same-shared data.

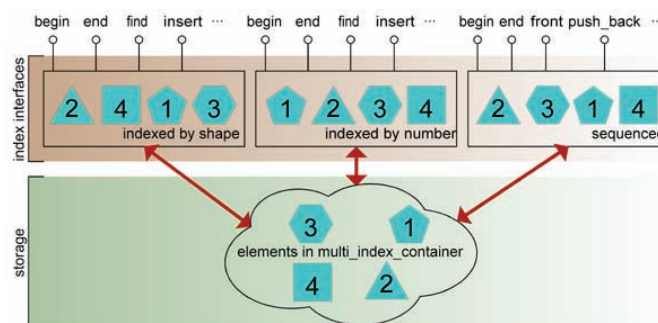


Figure 5.26: Concept of a multi-view scene representation.

This design does not only allow a perfect balancing between the 3D rendering and semantics view, but it also offers possibilities to optimize every single component. For instance, 3D rendering optimizations are often platform specific: some hardware may be more efficient to fetch different textures than to modify their lighting computation mode [Riguer2002, ATI-OPT2006]. By extending our multi-index container with additional views, we can adapt our system more easily. This is particularly important for controlling the simulation, as different semantic indexing may be necessary. They may represent more logical or intuitive views for accessing the objects could they be using unique ID or by classifying them in different categories (humans, dynamic objects, static objects). From a performance perspective, as our virtual worlds are populated with thousands of dynamic entities, it is important that we keep an efficient $O(1)$ access time on every representation. The container implementation is ensuring this requirement. Moreover, the spatial efficiency becomes acceptable on memory consumption as the number of indices grows. Finally, the memory fragmentation remains minimal especially in comparison to manual management, which can show up in more usable memory and better performance.

5.10 Architectural Patterns

An architectural pattern expresses a structure organization for software system. It specifies system components responsibilities and includes generic rules and guidelines for managing component collaborations. [Buschmann1996] explain how software architectures can be driven by patterns. They illustrate the usage of patterns to describe and document large-scale software. Architectural patterns combine individual patterns into heterogeneous structures to facilitate a constructive development of software systems. Similar to the classical design patterns, many categories exist, such as patterns for distributed, interactive, or adaptive systems. [Simpson1999] goes further toward 3DRTS by describing a set of patterns designed for 3DRTS developments.

5.10.1 Concurrent Design Patterns

The well-established design patterns for OOP do provide strategies for solving recurring design problems with a systematic and general approach. Many similitudes with the domain cover by parallel computing are emerging. Both use similar structures for controlling the communication behaviors with specificities such as deadlocks and data synchronization [Schimdt2000]. Starting from this observation, [Goswami2001] propose parallel architectures skeletons. The idea is to provide high-level architectural-skeletons as reusable components for common parallel computational patterns. [Anvik2002] apply the Design Pattern methodology for concurrent systems. They describe the CO_2P_3S (*Correct Object Oriented Pattern-based Parallel Programming System*) system, that generates parallel programs from parallel design patterns. They demonstrate that CO_2P_3S can generate structurally correct parallel programs with good speedups on shared-memory computers. [Mattson2004] approach was to develop a pattern language allowing focusing on concurrent algorithms rather than on hardware specific. They develop the *PLPP* (*Pattern Language for Parallel Programming*), embodies a development methodology in which parallel applications are developed around a sequence of dedicated patterns, ending with code. The main idea is to decompose concurrent software into a layered architecture:

- A pattern language for parallel programming.
- A component-based framework.
- Low-level portable APIs for parallel computing.

5.10.2 Concurrent Micro-Kernel

This pattern provides to each controller a chance to execute tasks, by dispatching the workflow over threads. Without this pattern, components objects are typically update by a set of hardwired controller functions called from a main loop, such as depict in the following listing:

Listing: systemUpdate

```
void systemUpdate(float fElapsedTime)
{
    ComponentsVector::iterator it(vecComponent.begin());
    ComponentsVector::const_iterator itEnd(vecComponent.end());
    for (; it != itEnd; ++it)
    {
        (*it)->update(fElapsedTime);
    }
}
```

This updating style force the components update to be synchronized on the same frequency. In some circumstance, some components require a higher priority. Thus, an alternative is to extent the concept of micro-kernel to dispatch the workflow more accurately. Different policies can be proposed to give to each controller a chance to run on different threads and frequencies. One solution consists to separate the list of components into sub lists that are dedicated to run on one thread. Alternatively, for systems where most components can run at the same frequency, OpenMP primitives can be used as presented in the next listing.

Listing: Micro-Kernel using OpenMP primitives

```
void systemUpdate(float fElapsedTime)
{
    const int iSize(vecComponent.size());
    #pragma omp parallel for schedule(dynamic)
    for (int i(0); i < iSize; ++i)
    {
        vecComponent.at(i)->update(fElapsedTime);
    }
}
```

In this example, the scheduler dispatch component updates on different threads. As each component update has a different processing usage, the embedded dynamic scheduler provide by the OpenMP standard allow to spread the workflow on demand. In other words, as soon as one thread has computed a single iteration, the scheduler will provide the thread with another component update. This heuristic is particularly useful for scheduling unpredictable work. In effect, component's update can vary on a frame basis. By applying a dynamic scheduling, the system can respond to variances with better accuracy. The model of concurrent multitasking dispatcher offer good scalability for different platform as OpenMP dispatch the workflow base on the hardware specifications. This mini-kernel automatically map to the number of hardware threads. This is particularly useful when the system is intended to run on different configurations (single processor, dual-core, dual processors...).

5.11 Multithread Models

Hardware platforms for 3DRTS do not longer rely on single processor unit. The technological advance from single processor to multiprocessors units changes the programming model. New methods focusing on concurrency have to be considered to fully take advantages of the additional CPUs. Many concepts developed for intensive computations such as the Flynn's classification, task scheduling and dependency graph as well as the Bernstein condition to concurrency models have to be adopted by 3DRTS programmers. The traditional approach consist to rely on a global loop that is responsible to perform the necessary computations updates such as rendering a new frame or computing animations and AI. Some systems are trying to slightly change this configuration by allowing updating some components at a lower frequency, but the principal scheme is remaining an iterative process. The current trends in hardware technology as well as the subsequent increase of complexity force the emergence of new paradigms, which will transform single threaded programs obsolete [Balfour2006, Wilson2006]. To maximize the processors throughput, developers must massively parallelise simulation components. Naturally, the parallelism cannot be limited to low-grain parallelism that affects only low-level elements such as math functions or I/O routines. The introduction of multithreading (*SMT*) processors

for PCs such as the Intel Pentium 4 have begun to widespread the installed base of platforms which benefits from parallelization. The next generation of home consoles enforce this trend by adopting multi-core architectures. As an outcome, multithreaded code for 3DRTS is promoted beyond the scope of general PCs and Workstations computers. Systems architectures that will continue to rely on a single threaded approach will be discarded as soon as the propagation of multiprocessors becomes the predominant architecture for building 3DRTS.

5.11.1 CPU and GPU Analogy

Most recent 3DRTS take advantage of the high parallelism processing power offered by modern GPUs [Harris2006]. The Figure 5.27 illustrates that even single threaded applications benefit from parallel computations. The distinction with multithreaded applications is that CPUs can run at full speed using all the hardware threads available. This leads to additional level of performance and ability to provide more energy to non-graphical elements [Owens2005].

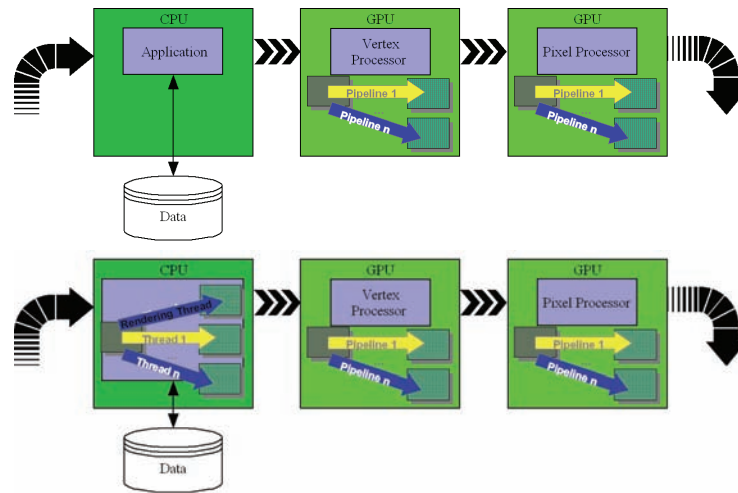


Figure 5.27: On the top, a single threaded application using the parallel processing power of modern GPU. On the bottom, a multithreaded application, that dedicated one thread for the CPU/GPU communication.

5.11.2 Concurrency Design

The nature of concurrency for 3DRTS requires to carefully understanding the different hardware components and their interactions with each other. For instance, the existence of specialized coprocessors such as GPUs is dedicated for specific tasks that define major elements of the dependency graphs. These prerequisites guide the design of a task scheduler that spread the workflow among the hardware coprocessors. In addition, it run general purpose or simulation code in the main CPU. The difficulty is to design a system that can reduce the dependency graph for 3DRTS, which generally have important amount of data running into multiprocessors architectures. To handle all theses mechanisms, engineering methodologies need to be apply such as synchronization primitives [Hoare1974] (monitors, locks, semaphores, threads pool, threads worker, reader/writer lock...). The sequential approach comes with severe limitations that do not scale up with multiprocessors architectures. By profiling the system with dedicated tools like VTune [Intel-VTune], we can observe that some components may be I/O bound or have to wait for some driver acknowledgments before processing further. Thus, it becomes important to decompose the system into sections that can be combined and executed in parallel. For instance, the principle of Bernstein [Bernstein1996] should guide the developers by ensuring that the workflow minimize the overhead involved by its proposal conditions, which define that two separate processes must be flow independent whenever it is possible.

5.11.3 Designing with Regards to the Dependency Graph

Keeping the control on objects interconnections at the components level is necessary to keep good potential to spread components workflow over several threads. [Gold2004] describe such objects dependencies as “the curse of Hades”. [McConnell2004] goes further by describing that dependency graphs are the principal bottleneck affecting software performance. One typical issue is the system is forced to take components B, C, D,

if you want to use component A, as component A was design to rely on the existence of the others components. This happened with components enhancing graphic rendering, which need the 3D graphic renderer. This is particularly problematic with C++ code base, as C++ dependencies are transitive. Another situation is to reduce code sections that are shared between modules. For instance, controlling virtual humans combine several elements. The characters need to be rendered on the screen using 3D graphics methods, while an animation module generates characters movement gestures. This means that during the rendering of one character, the renderer must not read the skeleton matrices simultaneously than the animation module is computing a new frame. This implies that the processing of both the animation and rendering modules should ideally not perform simultaneously or at least that at a given time, the rendering and animation update of one particular character are not proceed altogether. A strong technique to reduce the dependencies is to localized them in specific libraries or plug-ins that encapsulate them in a tightly defined unit. There are various ways to implement this in an object-oriented way; design patterns such as Shield, Bridge, and Adapter are good candidates [Gamma1995].

5.12 Performance and Optimizations

During the development of massive scale software, it may become difficult to clearly identify where the performance bottlenecks are. Many times, developers think that part of their system need to be optimized which end-up with minimal performance impacts due to the interconnections of components. The measurements of systems performance and the resulting optimizations is a critical process. Optimized code tends to become less reusable and have a limited clarity. This process is also a time-consuming task and reduces the code robustness by introducing programming tricks that may not work perfectly in all situations. Even, so it may be tempting to optimize early in the development process, developers should try to concentrate on improving time-critical code sections, rather than optimizing unnecessary large code base. In many occurrences, the good usage of programming languages keywords such as *const* or *restrict*, and proper compiler flags may optimize the code with limited manual interactions [Isensee2001, Meyers2001, Dewhurst2002, Bartolomeo2003, Ericson2003, Meyers2005, Stroustrup2005, Meyers2006, Stroustrup2006]. Generally, inner loops optimizations have less impact on performance than high-level optimizations. The use of profilers helps to identify bottlenecks with accurate statistics. In effect, developers tend to be not proficient when guessing for performance bottlenecks. Profiler tools such as Intel VTune [Intel-VTune] runs the application and measure time spend in routines and collect CPUs and memory usage. Their measurements serve to extract time-critical code sections. It will also help to discover some direct influence of high-level design with systems performance such as the cost of virtual functions [Isensee2004, Thomason2005]. They allow for dynamic binding in C++, but the inner mechanism relies on looking up function pointer in the virtual function table of the considered calls. On system with limited cache memory such as the Sony Playstation 2, the impact of designing systems architectures, which use intensively virtual functions, is not negligible [Llopis-C++2003, Whigham2005]. Thus, it is better to consider keeping virtual functions at high levels of granularity. This implies to perform virtual functions on a per-object basis rather than on internal loop functions. C++ is a powerful language that offers many intrinsic mechanisms while still allowing accessing directly the hardware [Lippman2005]. One difficulty when programming in C++ is that some intrinsic behaviors may be hidden due to the encapsulations that influence the run-time performance [Meyers-Perf1994, Meyers1994, Isensee-GDC2006, Lee2006]. This include wrong usage of the STL library, intensive function calls, hidden allocations, and dead code [Meyers-EffecSTL2001, Meyers-STL2001, Llopis-C++2003, Isensee2004, Freeman-Hargis2006]. Some examples include the use of STL containers hiding the copy of items and memory allocations. Another situation occurs with type conversion casts or passing function arguments by value. In addition, the allocation of array should be avoided during the simulation loop, as there may penalized deeply the system performance. Another myth with programming optimizations is related to promote assembly code. Assembly is not portable and is difficult to handle properly. It is not always true, that a function written in assembly performs better than its C counterpart does. Many times, developers will produce assembly code that is less performing than optimized compiled C code [Abrash1994]. In effect, current set of compilers are proficient at producing optimized code. Assembly code should be adopted only on very specific and low-level functions whenever the compiler do not handle the code generation properly on its own.

5.13 Conclusion

The system performance was measured with concrete implementations. This gave us information about the system limits. However, some limitation factors are the results of relying on modules not following the concept described here. Thus, in the next chapters, we will illustrate a complete vertical implementation of components helping to understand how low-level and cache coherent libraries can be integrated into a more flexible and high-level hierarchy.

Chapter 6

System and Technology

In the scope of this thesis, the goal is to promote methods and design patterns for extending the usability of 3DRTS systems for non-programmers. Despite, the ideology to presents fully generic solutions, the system need to rely on efficient low-level modules. In this chapter, we will analyze the current trend in computers hardware and their impacts on the overall design. This implies the importance of multitasking workflow and their interactions with previous approaches. This will lead to describe some low-level optimizations connected to time-critical code sections of our system. In effect, CBD have the reputation to over-design software architecture, which in turn reduces the run-time performance. If CBD increase the components encapsulations around a multi-level representation and high modularity, the real performance impact is not as important, as initially thought. In fact, the highest levels of the hierarchy are consuming a small percentage of the overall processing power, following the 80/20 rule [Koch2000], where a small code base is responsible for the majority of the resources usage. Therefore, most of the system performance relies directly on low-level routines. If we can optimize these 20% of the code whatever we do in the 80% remaining is not affecting too deeply the overall performance. As an outcome, CBD need to rely on efficient low-level routines for validating the method. The goal is to extract time-critical code sections that require intensive optimizations. This includes handling the specific constraints related to system management in a concurrent world, forcing a paradigm change in the conception of algorithms [Miller2005]. Developers need to deal with alternative approaches and to re-think some classical computer graphics algorithms. In particular, techniques reducing the memory traffic like cache-oblivious algorithms or that massively parallelizes the processing of large datasets are important. In addition, the recent progress made with compiler optimizations ease the vectorialization of code sections. This consist to employ *C99* and *C++0X* keywords and parameters such as *restrict*, *inline*, *floating point precision*, *loop unrolling*, *const...* In this chapter, we will illustrate how to design the system so it can be decoupled in light and stand-alone components, which could eventually run on a distinct thread. This is important to spread the workflow on more threads when needed. Next generation of computers hardware will increase the number of hardware threads available and will differ greatly from each other. Some problems that are ignored by single-threaded applications become critical in concurrent systems. These include the methods that ensure consistent access on data as well as providing the abilities to shared caches among processes. Unfortunately, the high-level of parameters that may interfere in system performance oblige to rely on specific tools such as VTune [Intel-VTune], that highlight code sections where an alternate load balancing could be applied.

6.1 CPU

The developments and improvements made with computer hardware and especially CPUs over the last decades were important. Processors manufacturers [AMD, IBM, Intel, Sun], have followed a similar path of optimizing performance. Rather than focusing entirely on the hardware design, the major gains were done using better micron process technologies, which is associated with lower power consumptions and faster frequencies [Moore2002]. A well-known example is the Pentium 4 line of processors and their very long instructions pipeline has reached its limit. Moreover, today processors are reaching their chemical properties constraints. Therefore, alternatives approaches have to be elaborated for increasing processing power. The processors manufacturers are doing the transition from a vertical to horizontal methodology. They are incorporating into their architectures hyper-threading and multi-core technology. The mass production of multi-core processors started in 2005. PowerPC and X86 processors already use this technology and manufacturers roadmaps describe their interests for concurrency architectures designs. Multi-core technology is now widely available within PowerPC and X86 processors and is becoming the major theme of improving performance [Krewell2004]. From a software perspective, the software design approaches within a concurrent context, will change software developments similarly to the introduction of Object Oriented design. Software applications that will not be designed to take advantages of concurrency models will not benefit from the previous vertical extra speed improvements of processors. During the past decades, most software applications were running faster on computers that are more recent. This vertical gain in performance is often refer as the “*free performance lunch*” [Sutter2005]. This had the advantages of increasing the software complexity following the same paths as their predecessors. The idea that CPU clock speed was the principal measure of performance was misleading developers and customers’ altogether. Many people are convinced that a CPU running twice the speed of

another one involves also twice the performance. Naturally, a processor is only one element of measurements when analyzing computer performance; such elements like memory cache size, memory bandwidth also contribute to the system capabilities. Whatever, software developers were able to rely on the constant linear processing power improvements. Thus, in situations where the software requirements were too high for the current hardware, waiting a couple of months would see the exact same software run smoothly due to the additional processing throughput.

6.1.1 Multi-Processors Architectures

Multiprocessors architectures for concurrent programming models have been developed over the years for software applications not related to 3DRTS. For instance, the Flynn's classifications encompassed all computers systems with a single processor executing a single stream of instructions on a single stream of data [Flynn1966]. He classified such system as *single instructions stream- single data stream* (SISD). Single processor computers running a sequential program fall under this category. When processors execute the same set of instructions on different stream of data are termed *single instructions stream – multiple data stream* (SIMD) computers. Another classification cover computation of *multiple instructions stream – multiple data stream* (MIMD) is assigned to systems where independent processors execute programs which work on different stream of data. This is related to systems where different instructions are running in parallel. The last category covers the ability to split multiple instruction streams over a single data stream. [Flynn1966] signification is termed *multiple instruction streams - single data stream* (MISD). The usage of the different techniques also rely on the hardware architectures which may come with shared Memory [Dahlgren1999], or Distributed Memory as well as vector computing as used within the Sony Playstation 2 hardware [SCEE2000]. [Costa2004] demonstrate that the most appropriate environment for 3DRTS relies on platforms using shared memory and vector processors. From a design perspective, the principal purpose of multithreading environments is to clearly identify the dependency graph for optimizing the set of instructions that can be executed simultaneously on the same or different datasets. This allows exploiting the potential gain over a sequential and single-threaded approach. The development of parallel code paths required to dedicate some attention to the tasks decompositions. [Grama2003] describe the different steps of parallel computing from the hardware consideration up to the software analysis. They focus on describing the parallel programming principles that cover the threads communications and data synchronization primitives as well as covering threads API and directives. The gap between the theoretical and practical performance of parallel computing is important. Ideally, a parallel program is composed mostly by independent tasks. In practice, these situations are uncommon. Many tasks have cross-dependencies on data and need to communicate with each other. To ensure efficient parallelism, it is important to understand and generate a dependency graph that will represent the intercommunications between tasks. By analyzing the different configuration of tasks dependencies for a given simulation, the system can be optimized for executing tasks in specific order. This requires identifying the problematic related to parallelism programming schemes, by observing the tasks dependency graph and by clearly specifying tasks that can be executed simultaneously. This process schedules the tasks execution ordering, reducing the situations where a CPU waits for processing datasets. Different types of dependences between instructions sets exist, reducing opportunities to run code in parallel. The data dependencies can be categorized into the following group, which represent the different forms of dependence that affect data processing and parallelism code [Hwang1993]:

- *Flow Dependence*: represent the influence of statements when the execution path of one output result from one statement serves as an input for another statement.
- *Shared Memory Dependence*: represent memory location that will be access and altered by different statements. The main limitation with shared memory is that whenever one cache is updated with information that can be used by other processors, it needs to update the shared memory immediately.
- *I/O Dependence*: occurs when several statements need to access the same I/O devices.

Once a program can be decoupled from a sequential process into smaller independent components, it can be analyzed and adapted to the aforementioned methods. By analyzing the data and resources dependencies, developers can configure their systems to optimize the opportunities for parallelism.

6.1.2 Current Limitations

Already many applications and especially 3DRTS are throttles with different set of issues along CPU throughput. The improvements made with memory bandwidth and memory sizes are one of the principal

systems bottlenecks [Kaspersky2003]. The main problematic is keeping the processors busy. It is common that significant CPU cycles are spent on I/O operations or data acknowledgments [Davies-MT2005]. On this regards, the concurrency models do not improve the situation, as faults pages and cache misses are increasing even at a faster rate [Handy1998]. For optimizing the workflow, today processors are trying to provide a set of functionalities, which include long instructions pipeline, branch predictions, as well as executing multiple data in the same cycle. Some processors called out of order CPUs can reorder the stream of data on the fly. All these techniques are tentative to exploit as much as possible CPUs processing throughput by reducing the shortcomings of memory latency and sub optimized code. In some circumstances, those optimizations are affecting the code execution paths, which may end-up altering the programmer's expectations. The consequence is that some code designed before the apparition of such optimizations may not work properly on recent computers [Intel2005]. This shows that processors manufacturers have reached some limits with past architectures. The fundamental changes of adapting the concurrency model have a distinct impact on software developments.

6.1.3 Technology Trends

In the year 2005, the processors hitting the shelves for workstations and embedded systems are design with concurrency in mind. The future performance improvements in processing power will be the combination of technologies like hyper-threading or multi-core CPUs, and better cache managements [De Gelas-PI2005, Olukotun2005]. The combinations of these techniques are the key factors in the near term CPUs developments [Binstock2003]. In the year 2004, Intel has introduced hyper-threading technology into their Pentium 4 line of processors. Apart the marketing advantages of such technology, hyper-threading really improve the performance. The idea is to run instructions in parallel using two hardware threads. With the Pentium 4 implementation, every hardware thread has its own dedicated level-1 cache while sharing the level-2 cache, the FPU, and integer math units. The performance boost cited is around 5-20%. On the other side, multi-core is to set several CPUs on a single ship. With this approach, the performance gain is about the same as standard dual CPUs system. If two threads are working on the same data set, the shared cache memory of hyper-threading may perform better than running them on different cores. Thus, some processors are combining both technologies. The major difficulty for the developers and compiler is to spread the workflow on all the hardware threads cautiously. Those technical advances are beneficial only for massively multithreaded applications. Due to their complexity, multithreading implementations are not well spread over the developers' community. The technical issues like data synchronization and the lost of iterative execution flow prevent the use of threads in many software architectures. In fact, both languages and tools have to evolve for helping this transition. The impact is going above the threading management. Extending the number of cores and hardware threads available will not induce that the processing power will increase on the same scale, as memory evolution is not following the same improvements. Thus, successful software architectures will be the one that can provide different layers of abstractions capable of running in a concurrent world, while keeping intuitive and flexible high-level perspectives. The main techniques used during the last decades for software complexity developments have to be rethink As developers cannot anymore hoping for an automatic improvement of their software with time. Certainly, concurrent programming is already use by few developers. Similar to the OOP model used in the late 1960 with programming languages like Simula [Dahl1997] and Smalltalk [Kay1993], the mainstream explosion of this methodology took place only in the 1990 and the constant growth in large-scale software developments. Therefore, even so concurrent programming is not new on itself, it has not reached mainstream developments. The hardware technology is forcing developers to apply the concurrency model for keeping their software competitive. The modifications involved by mainstream concurrency software will deeply change software engineering practices, which will have to master the added complexity and important learning curve.

6.1.4 Benefits and Costs

Some advantages of the concurrency models allow separating independent flow of data into separate threads. Another clear benefit is that it prevents that the application becoming too dependants on the latency of I/O operations affecting different flows. For instance, single-threaded applications, and especially 3DRTS are very sensitive to I/O operations. Thus, instead of blocking the whole application on such occurrences, multithreading systems are more likely able to keep the processors busy. Developing concurrent applications has a major impact for any software. To ensure that the data is consistent, threads need to acquire locks on them. Also, if the software used more threads than the number of hardware threads, then many CPU cycles are spent during context threads transitions, which may end up minimizing the benefits of running several tasks in parallel. Moreover due to the restricted number of tools for concurrent programming, the development costs are

rising. For instance, reproducing bugs within a multithreaded environment can be a real nightmare. Hopefully, we can see that tools providers like the one from Intel [Intel-VTune] or Metrowerk [Metrowerk-Analysis] are good steps toward mature tools. Moreover, the load balancing of concurrent flows is not trivial and differs for each application. Finding and correcting racing conditions is a difficult operation. The technical knowledge for writing concurrent code that is completely safe and efficient is important.

6.1.5 Performance Gain

The performances gains achieved with concurrent systems are limited by certain factors. One of them is the serial nature of some tasks. A subset of tasks will always need to be executed in specific order. This limitation is known as the Amdahl's Law [Amdahl1967]. The theory behind this law is that if 30% percent of an application is inherently serial, parallelizing the remaining 70% yields to a maximum theoretical speed-up. Depending on the overall number of processors, the speedup can be as low as 35% on two processors or as high as 70% on n processors (see Equation 6.1).

$$gain = \frac{T_{total} - (T_{serial} + \frac{T_{parallel}(1)}{Nb\ Pr ocessors})}{T_{total}} \quad (6.1)$$

Naturally, this scheme assumes the overall system relies on inherently serial code. The speedup performance can dramatically improve by decoupling the execution flow into separate components allowing avoiding unused processors. For instance, [Shi1996] illustrate the speedup discussed in [Amdahl1967] and [Gustafson1988] are in fact equivalent. The speedup factor can be interpreted as a function of the number of processors, but is also a function (implicitly) of the problem size. The Equation 6.2 illustrates the speedup function.

$$Speedup = \frac{T_{serial} + T_{parallel}(1)}{T_{serial} + \frac{T_{parallel}(1)}{Nb\ Pr ocessors}} \quad (6.2)$$

As we can see from Equation 6.3, if the number of processors tends to infinity, the optimum speedup factor is considerably limited by the code that remains serial.

$$OptimumSpeedup = \frac{T_{serial} + T_{parallel}(1)}{T_{serial}} \quad (6.3)$$

The Figure 6.1 shows the correlation of performance speedup in multiprocessors hardware is directly limited by the serial code [Miller2005].

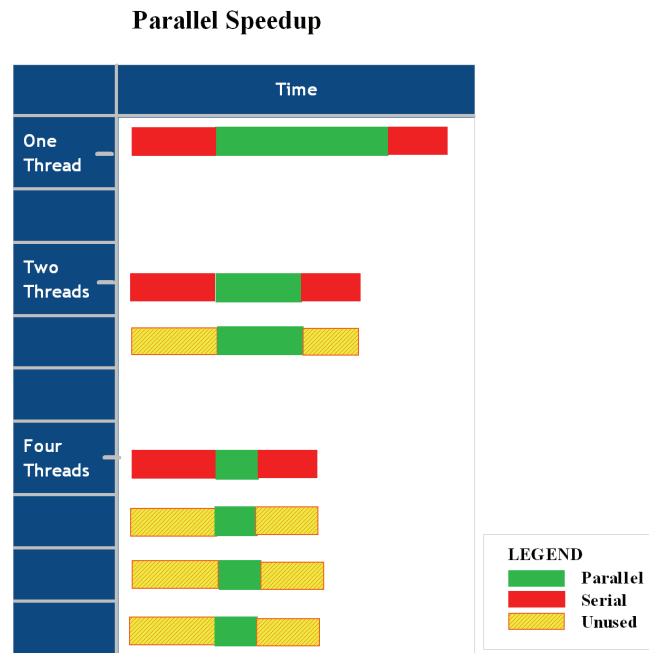


Figure 6.1: Serial code impact in parallel speedup values.

6.1.6 Tools

The shortcomings of current programming languages and the relative lack of experience from software developers with concurrent models require better set of tools. Programming languages as well as systems will increasingly be forced to follow the hardware trends and will have to provide primitives and tools for handling concurrency natively. Recent programming languages, such as Java or C#, include some support for concurrency, but some limitations that are difficult to correct, still prevent programmers to work efficiently and correctly using native functions. For 3DRTS, which are mainly developed using the C++ programming language, the situation is even worse. For many years, C++ has been used to write multithreading systems, but no standardizations were defined. For instance, the ISO C++ standard in its latest iteration does not even mention threads [Kaley1999]. This implied that multi-threading applications developed in C++ is based using platform-specific features and libraries and may come with incomplete features on some platforms. So far, only some initiatives try to promote some concurrency standards such as POSIX threads and OpenMP. They try to emphasis on the ability to produce code for implicit and explicit parallelization. The topic of resolving concurrency problematic, such a lock-based programming systems, remains an intensive and difficult research field [Sutter-PDC2005]. Developers working with concurrency are desperately calling for higher-level programming models moving away from the current state of today languages [Intel_MT2003, Sweeney2006].

6.2 Technology Awareness

Mission-critical software performance is influenced by the disposal technology. Hardware vendors rule the general conceptual design behind 3DRTS systems architectures developments. In effect, systems architects are forced to embrace new technologies for ensuring real-time performance. With the expansion of multi-processors hardware, it becomes important to clearly understand the underlined advantages and constraints they offer [Singh2003, Andrews2005, De Gelas-PII2005, Davies2006, Plumb2006].

6.2.1 Concurrent Models

With the apparition of computers capable of running several threads in one clock cycle, the concurrency model is becoming an important factor for taking benefits of current hardware processing power. The concurrency model has several layers and related terms. In some circumstances, the solution is to apply fine-grain threading models for improving the computation time of specific routines, while in other cases, a much higher level of abstractions and separations of functionalities is required. The notion of concurrent software overlaps terminology like multithreading, hyper-threading, or multi-core. CPU hardware comes with specific features such as symmetrical or asymmetrical cores. To optimize the performance, a concurrent system architecture need to understand the specificity related to the different models. The trend from both hardware and software perspectives to transform the conceptual single-threaded and sequential methodology to parallel processing are the results of the way computers hardware and software work. Even intensive applications like 3DRTS are not completely using the entire (and theoretical) processing throughput. In many situations, programs need to wait for certain events to happen. We can think of situations where the application is waiting on I/O operations acknowledgments or that the system needs to wait for synchronizing the 3D rendering with the monitor vertical screen refresh. With time, the wasted CPU cycles that affect single-threaded applications were analyzed. As for single-threaded software, there is no real solution to avoid these limitations, software and hardware architects were trying to promote new programming paradigms. Their solution was to allow several processes to run at the same time. On mono-threaded computers, OS will periodically switch back and forth between processes. This would allow letting one process do some computation while the second would be blocked. On modern computers that featured more than one hardware thread, several processes can run simultaneously avoiding the overhead of threads context transitions.

6.2.2 Multi Core Systems

Multi-core systems are piece of hardware that contains several processors on the same chips. The advantages of multiprocessing system allow running several threads on distinct CPUs, accelerating concurrent systems architectures. Well-balanced systems may also reduce the penalty of having to share single processor resources with the assumption that the number of threads matches the number of hardware threads. The developments of multi-core architectures are composed of two different categories. The first category covers symmetrical cores. The idea is to provide similar processors. The advantage of this approach is that one thread

would run similarly, whatever the core he is running on. This allows spreading the workflow easily. Symmetrical cores are the mainstream designs as they facilitate the concurrency model. Current X86 processors from both Intel and AMD features symmetrical cores. On dedicated hardware for home consoles, such as the Microsoft Xbox 360, the CPU refers as the xCPU can be described as a derivative of the PowerPC processors with three symmetrical cores. On the other hand, sometime at a cost of additional workflow balancing management, hardware architectures can opt for an asymmetrical approach. On such systems, different processors units are combined with some of them being dedicated to very specific tasks, such as floating points operations. Such systems intend to be used for 3DRTS is the CELL processor. The technical approach is to provide a generic processor associated with several co-processors dedicated for optimizing specific tasks. The asymmetrical configuration is much more complex to handle but excellent performance may be obtain when proper workflow balancing can be achieved [Calver2005]. More details about the specificities of CPU architectures can be found in the Appendix B. Depending on the threading models, the management of concurrent systems always faces similar troubles on different levels. For instance, at some points in the program, concurrent threads will have to share the same resources. Consequently, some resources will have to be protected from multiple accesses at a time. Not ensuring this requirement will produce inconsistent data. Another difficulty comes when two threads need to communicate. Depending on their locations, some data will have to be copied into several cache memories. This increases the probabilities of cache misses that may affect performance deeply. All those penalties are directly affecting the way concurrent models have to be designed. Sometime, it may be more efficient to have multiple copies of data to prevent the need of waiting for acquiring locked resources. When designing concurrent systems, it is primordial to see the system from both low and high-level perspectives. From a low-level perspective, algorithms that ensure good caches coherences will perform better on multithreaded systems. On the higher level, being able to separate distinct modules that shared minimal resources is also necessary. The complexity comes to find the good proportion between duplicating data with data synchronization and memory footprint. Generally, the management of shared resources is the principal factor preventing the scalability of multithreading systems, as it is close to impossible to design a system without common set of resources. Since multithreaded applications may have issues that are difficult to reproduce, developers need to rely on tools that profile the system. Creating an application that spread many threads is a relatively simple operation. However, to prevent that the system is suffering from threads communications bottlenecks require more dedications. The interactions and shared resources may delay the processing of data. This occurs when threads lock resources longer than necessary and are responsible for major system overheads. This leads to situations where each additional processor increase the OS overhead, which result that new processors contribute less and less to the overall system performance.

6.2.3 Multi Cores Systems and OS

Multi core systems are true multi processing computers. Every core within the system can be seen as a single processor. This means that those systems have the ability to execute several flows of instructions in parallel at each clock. In contrary to Hyper-Threading technology, multi-core systems are close from dual CPUs systems. They can run several threads at full speed as they can rely on dedicated execution units. Some systems have adopted both technologies where every core contains more than one hardware thread. The drawback is that some OS cannot make the distinction between these two kinds of hardware threads technology, which results on poor load balancing. Occasionally, it may happen that an application using two threads running on a dual core CPU with Hyper-Threading technology ignore completely one core.

6.2.4 Threading API

Programming concurrent software in C++, require the use of low-level mechanisms to compensate the weakness of C++ [Stroustrup1996]. The language was not designed with concurrency safety as a core feature and do not provide thread safety mechanisms on the class level. Moreover, threads management differs on each platform (POSIX or Windows Threads) [Carver2005, Kang2005]. To ease creating cross-platforms multithreading applications, developers need to rely on libraries that intend to provide thread interfaces for C++. For instance, [OpenThreads] and [JTC] are loosely modeled on the Java thread API. They generally support the management of the four fundamental types (Thread, Mutex, Barrier, and Condition). In our framework, we rely on the OpenThreads library for handling synchronization mechanisms by creating a base class (vhdObject), which is equipped with a mutex used for the scope locking design pattern. This pattern allows stand-alone scope synchronization, ensures appropriate lock, and unlocks operations, which is important in context of large-scale systems. The listing below highlights how synchronization mechanisms are hidden within the framework.

Listing Synchronization Methods:

```
void vhdMyObject::setValue( vhtReal rValue)
{
    /* monitor locking (only one thread may enter in this scope) */
    vhdSYNCHRONIZED(this);
    _rValue = rValue;
} //automatic unlocking of monitor (next thread may enter)

vhdMyObject::execute()
{
    {
        vhdSYNCHRONIZED(this); //only one thread at a time
        // ...
    }
    // ... parallel section
    {
        vhdSYNCHRONIZED(this); //only one thread at a time
        // ...
    }
}
```

Using the `vhdSYNCHRONIZED` macro gives the ability to secure access on data with a single line of code for all objects that inherit from the base class `vhdVoid`. As describe in the second example, code sections can be executed in parallel while ensuring that only one thread may alter the same data at a given point. Therefore combining the framework threading mechanisms with the usage of OpenMP pragmas, minimize the complexity of threading management.

6.2.5 Synchronicity

Concurrent systems need to handle the problematic of data synchronization. Several synchronizations methodologies can be applied [Henney2003]. Because, synchronization is tightly connected to the communication layers, direct dependencies of component are forbidden, restraining synchronizations mechanisms on components and lower levels.

6.2.5.1 Synchronization at Component Level

The highest level of synchronization occurs at the component level. The component level synchronization is often referred to the safe (or lazy) synchronization mechanism. In effect, it will prevent that multiple threads interact on the same internal component's data, by ensuring that only one thread can acquire the lock on this module. The advantage is that the developers do not worry about the internal dependency graph and use components as single threaded applications, as illustrate in the listing below. This approach eases developing concurrent software but come with an important performance impact, as the system cannot perform any update while changing internal object behaviors simultaneously.

Listing: Component-Level Synchronization

```
vhdMyService
{
    bool update( vhtReal rElapsedTime)
    {
        vhdSYNCHRONIZED(this);
    }
    //set and getters
    void setParam( vhtReal rParam)
    {
        vhdSYNCHRONIZED(this);
    }
};
```

6.2.5.2 Synchronization at Object Level

The object level synchronization represents the lowest level required by thread-safe components. This obliges a better knowledge from the developers, which need to understand the behaviors and dependency graph affecting the considered module to restrict synchronization mechanisms to the lower grain possible. This implies that objects have their own locking objects (monitor, mutex, etc.) which need to be acquired by threads whenever they need to access or modify some objects on demand. This allow to compute different tasks on the same component simultaneously such as updating the component while changing some objects parameters, but may provide significant performance overheads due to constant acquisitions of locks. In our system all objects that inherit from the class *vhdVoid* are directly equipped with synchronization mechanism at the object level.

6.2.5.3 Choosing a Synchronization Topology

Between these two extremes of synchronization mechanisms represented by the component and object synchronization levels, there is a need to choose an intermediate solution that allow parallel computation on the same component while keeping synchronization overheads at reasonable levels. The idea is to regroup similar objects that are likely to be computed altogether and to use unique locking objects. This can be applied to large group of objects that can be computed as a batch before moving on the next group [ATI-R2VB2006, Dudash2006]. For instance, a crowd is represented by a set of templates from which multiple instances are generated. As changing graphic states and uploading textures are relatively costly operations, this is likely that the rendering pipeline try to render all instances of one template at once, as showcase in the following listing:

Listing: By Group Synchronization

```
void update(vhtReal rElapsedTime)
{
    for (unsigned int i(0); i < uiNbTemplate; ++i)
    {
        vhdSYNCHRONIZED(pTemplate);
        const unsigned int uiTemplateSize(pTemplate->size());
        for (unsigned int j(0); j < uiTemplateSize; ++j)
        {
            // ...
        }
    }
}
```

Here, synchronization can be done on the template level rather than on the instance level reducing the synchronization overheads while allowing concurrent computation on that component. Naturally, when another thread wants to access an instance, the response time can be longer, since the component update may have to perform the entire current template instances, before unlocking the data structure. Thus, additional profiling is required to define precisely good synchronization levels to maximize performance. Unfortunately, the profiles may change dramatically from platform to platform depending of the intrinsic hardware such as the number of internal hardware threads and the general system workflow. As framework complexity evolve, it is often easier to write components to perform tasks as batch operation [Rollings2000].

6.2.6 OpenMP

[OpenMP] is an API supporting multi-platforms shared memory and multiprocessing programming in C/C++ and FORTRAN. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behaviors. In effect, modern C++ compilers like [ICL] and [MS_CL] can expend OpenMP macros. The strong elements of OpenMP are its intrinsic parameters that give portable, scalable model for programmers using a simple and flexible interface for developing parallel applications. The standard offers flexible schedulers that dispatch the workflow among hardware threads as depict in Figure 6.2. If OpenMP was primary used with messages parsing interfaces for programming on computer clusters, it become possible for 3DRTS developers to benefits from its architecture [Isensee2005].

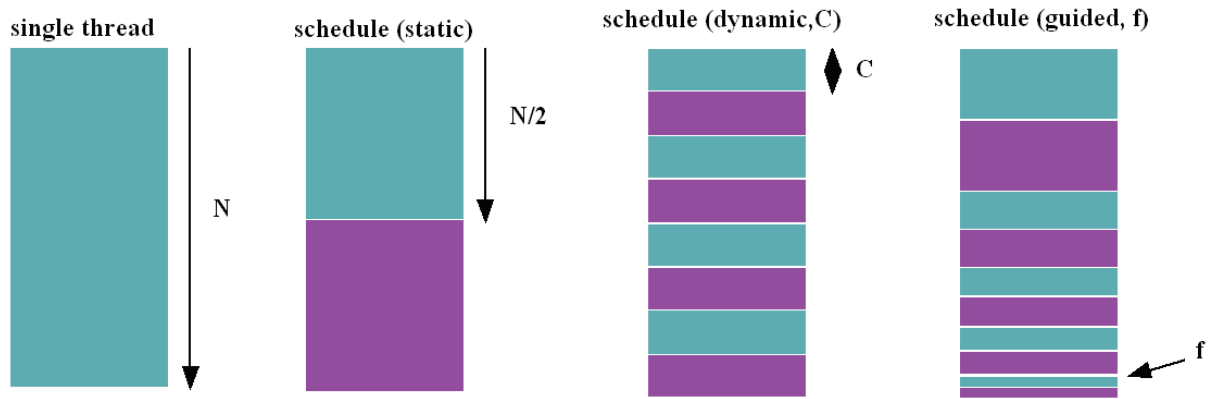


Figure 6.2: OpenMP task dispatching for two threads.

6.3 Cache Memory

Computer architectures use multi-level caches. A CPU cache is a layer of memory used by the CPU to reduce the average time to access information. The different cache levels found in modern computers is illustrated in Figure 6.3. Generally, each higher level of the hierarchy increases the memory size but is also considerably slower [Bhatti2003]. When the CPU wishes to read or write a location in main memory, it first checks whether the memory location is in the cache. If the CPU finds the memory location is in the cache, we say that a *cache hit* has occurred; otherwise, we speak of a *cache miss*. With a cache hit, the processor immediately reads or writes the data in the cache line. The proportion of accesses that result in a cache hit is known as the *hit rate*, and is a measure of the effectiveness of the cache. With a cache miss, most caches allocate a new entry, which comprises the tag just missed and a copy of the data from memory. The reference can then be applied to the new entry just as for a hit. Misses are slow because they require the data to be transferred from the main memory. This transfer incurs a delay since the main memory is much slower than the cache memory. The second issue is the fundamental tradeoff between cache latency and hit rate. Larger caches are both slower and have better-hit rates. Therefore, modern computer architectures present a multiple levels of caches.

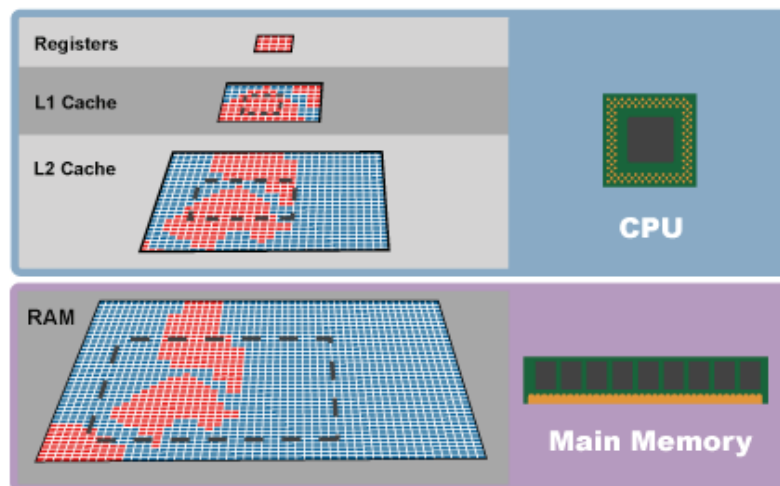


Figure 6.3: Memory cache layers attached to the CPU (Registers, L1 and L2 cache) and the main memory.

6.3.1 Cache Coherence

The different memory layers and their accessing time underline the importance of cache coherence for good performance. CPU consumes information provided to it by massive slow storage devices like hard drives

or optical disks drives. It uses information already stored in the main RAM, which act as producers for the CPU. The high performance of today CPUs and their faster evolutions than memory architectures have increase the gap between CPU processing throughput and memory bandwidth. The memory clock speeds do not provide enough bandwidth to keep the CPU running at full speed. In many situations, CPUs have to wait on data acquisitions [Goodmann1998]. Cache coherence methodology reduces memory and processor traffic. Creating algorithms with cache coherence is mind is difficult. Global systems architectures cannot apply these techniques on a global scale. Instead, they have to apply such techniques on lower level routines in their hierarchy to ensure that their workflow remain optimal. Fortunately, less than 10% of the code base is responsible for 90% of the processing workflow [Koch2000]. With such schemes, it becomes more important to design algorithms that perform well with caching than on pure RAW performance. The relative time to move the data from memory layer in comparison to processing time is growing deeply. To counter these side effects, computers architectures try adopting a multiple layers design for memory management. At every level, the memory size become bigger but the access and read time become slower. One might argue the solution would be to increase the memory size on the lower memory levels reducing the cache misses operations. Cache misses represent situations where some data is not located to the specific memory layer and need to be transferred from upper to lower memory layers. However, increasing any memory size also increase the seek time, which is the duration to fetch data from the memory. In addition, the price of low-level cache memory is prohibitive. Thus, hardware architects need to find the proper balance between cost, memory size and memory bandwidth.

6.3.2 Spatial and Time Locality

The notion of spatial locality cover labeling the generic rule assuming the memory that need to be access altogether are also stored altogether. The second assumption concern the time locality, which tell that the same chunk of data will likely be accessed again in the near future. The spatial locality coherence can be applied on all the different memory layers. On the lowest memory levels like the level-1 and level-2 caches, algorithms specifically designed with cache coherence like cache awareness and cache-oblivious algorithms are the best candidates. On a higher level, it consist to analyze the resources usage behaviors and to locate resources accessed at the same time (see Figure 6.4). This is important for optical disks storages where seek time is important. When storage costs are low, it may also be beneficial to duplicate data, especially for continuous worlds relying on constant data streaming. The spatial locality in code represents small data chunks batches by the CPU at a time. However, spatial locality requires avoiding unattended jumps and branches so that the CPU can execute larger contiguous blocks of data without interruptions. Fortunately, 3DRTS have decent spatial locality for code, as they often use a relative limited small amount of code operating on very large datasets.

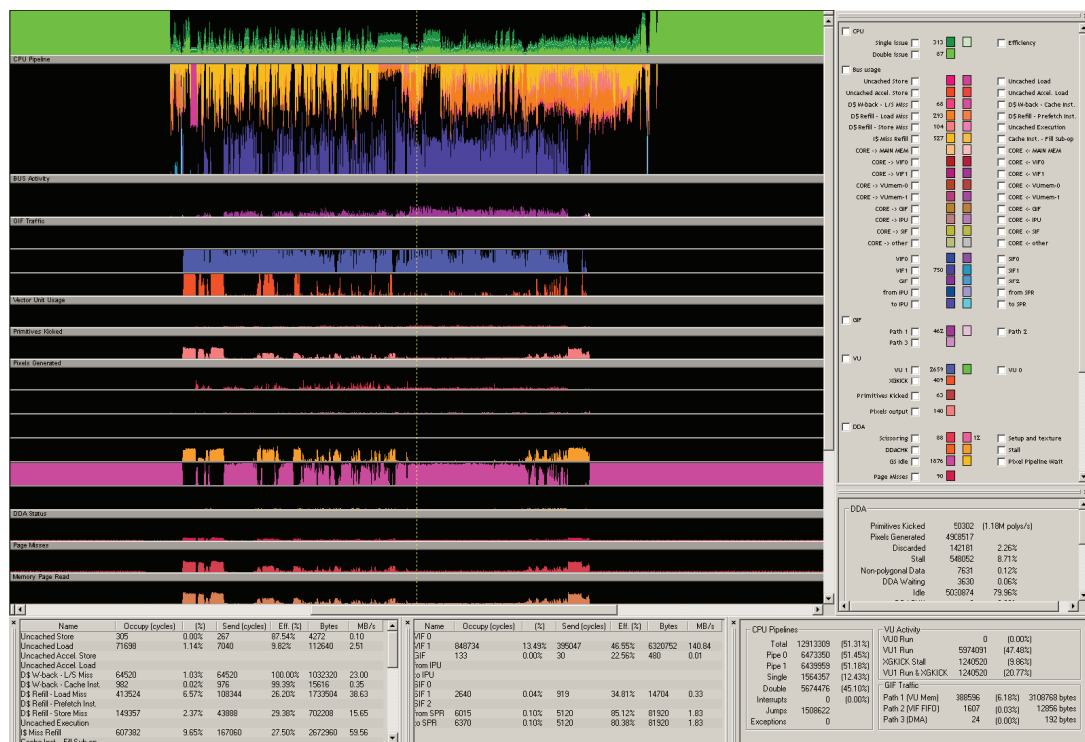


Figure 6.4: Performance Analyzer Tool [SCEE2002].

6.4.1 Non-Determinism Software

A strong advantage of single-threaded applications is that their workflow execution is deterministic. Mostly, they use a global loop that executes tasks in an interactive way. Thus, developers can understand and reproduce the simulations workflow and discover bugs more easily. In contrast, concurrent systems are clearly non-determinism. As the OS is managing threads transitions, some nasty bugs cannot always be reproduced. In particular, deadlocks and improperly synchronized data may not appear within the debugger due to different threads scheduling. Another aspect concern the system performance, where threads may stalls or waits as well as being affected by memory access issues such as race conditions. The transition from determinism to non-determinism software is a challenge. In fact, bad design or poor code that was not affecting single-threaded applications may become critical in a concurrent system.

6.4.2 Concurrency Specific Issues

The concurrency-programming model comes with its own issues related to threads management. The non-determinism aspect enforces the problematic by hiding or exposing situations that may change with minimal difference in the threads scheduling. The thread terminology includes:

- *Deadlock*: occurs when some threads are blocked to acquire resources held by other blocked threads.
- *Livelock*: occurs when all threads are blocked or are unable to proceed due to unavailability of resources and the non-existence of non-blocked threads allowing making those resources available.
- *Race Conditions*: occurs when two or more threads try to access the same object at the same time, and the behavior of the code changes depending on who wins.
- *Starvation*: occurs when one thread cannot access the CPU because one or more other threads are monopolizing the CPU. This may happen when the frequency or threads priority differs deeply, and that higher-priority threads do not yield control of the CPU from time to time.
- *Thrashing*: Happen when the number of active threads is considerably higher than the number of disposal hardware threads. The consequence is that the program makes small progress because of excessive thread context switching. In the worse situation, this may lead that no time is available to execute the code.
- *Death*: happen when a thread is throwing a non-catch exception or is terminated in the curse of its running context.
- *Leak*: occurs when resource are not reclaimed because a thread was created, but not run.

In the curse of our developments, some of these situations were observed. For instance, race conditions were appearing when more than one component was trying to modify directly the same ISO. However, the biggest problematic faced was due to threads starvation. In particular, we observed that the animation components required locking the virtual humans' skeletons. As our 3D rendering run in a separate and high-priority thread, the animation component was not able to access those resources whenever the 3D rendering was running at high frame rate (~90Hz). To resolve this issue, we were forced to yield the 3D thread at specific synchronization points to give a chance to the animation component to run. Another situation appear between our skinning component and the 3D rendering component when both were executed in parallel as up to 90% of the time was spent on lock acquisition. The solution was to force high-interconnected components to execute them in the same scheduler preventing parallel execution. Here, it is very important to rely on dedicated tools that highlight these problems such as [VTune2004], [Compurware], [AMD-CA] or [SN-Systems].

6.4.3 Modular Parallelism

Modular parallelism consists to separate the simulation code into self-contained packages such as components in CBD. Every package has to present a set commonality that interacts together with minimal intercommunications with the other packages. Different heuristics can be chosen. The modules can be executed on a dedicated thread and react as if it was a single-threaded system but may also benefit from fine-grain parallelism routines. Generally, components update involves executing a large amount of code but also minimize the intercommunications between threads, which are one principal bottleneck for concurrent systems. As the code execution is dispatched into separate threads, the system may choose to compute the components update at different frequencies. This allow to compute physics update on a high frequency for more accurate results but also to ensure that some components like the rendering or input components can run within

interactive frame rates. This help to provide a more consistent experience for end-users, and to dedicate specific components implementations taking advantages of platform specific functionalities. The Table 6.1 illustrates the different levels of parallelisms and the potential code sections candidates [Miller2005, Wall2006, Wu2006].

Table 6.1: Parallelism layers among technical constraints.

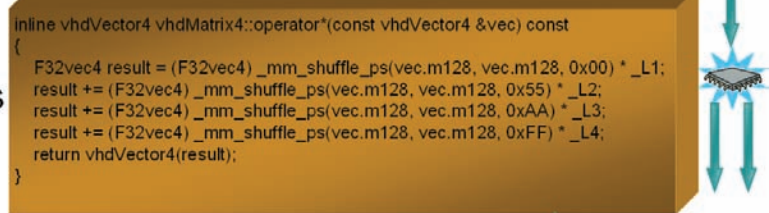
Constraints	Yes	No	Potential Candidates	Speed-up potential
<i>Intrinsic Fcts</i>	Fine-grain	Pipeline, Farm	Low-level routines	Size of registers
<i>High intercommunications</i>	Pipeline, fine-grain	Farm	Physics, Animation	Depends on the data decoupling
<i>Custom Memory</i>	Fine-grain, Farm	Pipeline	3D renderer, 3D sound renderer	Important
<i>Unify Memory</i>	Pipeline	-	Most code base	None on itself
<i>Execution order</i>	Fine-grain, Pipeline	Farm	Special effects (particle, clouds...)	Lowest denominator
<i>Asymmetrical Cores</i>	Fine-grain, Pipeline	Farm	Low-level libraries	Good for specialized code
<i>Symmetrical Cores</i>	Farm	Fine-grain, Pipeline	Components	Good for general purpose code

6.4.4 The Parallelism Categories

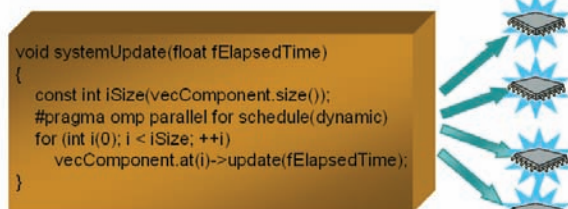
Depending on the level of parallelism desired by the architecture, different techniques and paradigms should be applied [Dawson2006]. The Figure 6.6 describes the three categories retained in our architecture. On the instructions level, we take benefit of SSE instructions for many heavy processing aspects such as math, physics, or skinning routines. The second categories represent the parallelization on the blocks level. At this middle layer, we rely on OpenMP due to its particularities and its great adaptation to a wide range of hardware. However, the OpenMP limitations such as preventing the selection of thread priorities prevent its usage on the components level. At this level, we need to rely on standard threads. In our situation, we rely on an open-source and cross-platforms thread API reflecting the Java Thread Interfaces [OpenThreads]. Using these three categories gives more flexibility to parallelize the system workflow adopting the most agile methods on lower levels and keeping the higher flexibility and granularity offers by standard threads APIs.

Parallelism Categories

- Instructions Level
 - SSE Instructions



- Blocks Level
 - OpenMP Pragma



- Components Level
 - Threads

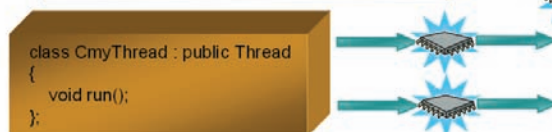


Figure 6.6: The different parallelism categories.

6.4.5 Micro Concurrent Model

The micro concurrent model consists to accelerate the code execution using different datasets. This implies that multiple threads execute simultaneously the same code. In this configuration, we apply parallelism directly on the data level. For example, particles systems or characters positions updates are good candidates for fine-grain parallelism, as data are isolated. This methodology is also more independent of the overall design and can be apply to low-level elements within existing single-threaded applications. Moreover, as each thread is fully independent, makes such algorithms very scalable. Some recent compilers can apply limited automatic instructions parallelization during the code generation process. When developers provide hints to the system, by carefully adopting languages keywords such as *const* or *restrict* in C++, fine-grain concurrent code may be generated. However, in many circumstances, the compiler is able to discover that specific data can be safely parallelized, resulting on an iterative computation, even so the logic could be computed in parallel. In effect, C++ is not well suited for concurrent programming and specifically for fine-grain concurrency model. Developers can either rely on architecture specific keywords that allow benefiting from intrinsic functionalities such as SSE instructions [Klimovitski2001]. To take advantages of cross-platforms fine-grain architectures, the [OpenMP] initiative was developed [Lee-OMP2003]. Its advantages are various: OpenMP rely on pragmas, which mean the same code base can compile on systems that do not support them. In addition, the design was oriented for optimal performance. OpenMP was developed more than 10 years ago. Then, the primary goal was to help the research community working on supercomputers systems. Modern compilers like Intel C++ Compiler 8.1+, GCC, and MS Visual Studio 2005 are all supporting the standard. The usage of OpenMP relies on directives that are activated through `#pragma`. Each of these directives allows defining parallelizable sections as well as workflow mechanisms [Gatlin2005a]. An additional restricted API is also available allowing obtaining information on the system like the number of hardware threads available as well as providing synchronization primitives. This API allows to restraint the parallelization on a sub-set of all the hardware threads [Gerber2003]. OpenMP allows to define few and well-defined synchronization points. Some performance measurements made by [Isensee2005, 2006] have shown that same code develop using OpenMP or by using a native thread API, reveal that the performance are similar without to worry about platform specific code. However, the usage of OpenMP is limited to simple loops and parallel code sections. All the operations have to be order-independent. In addition, OpenMP is generally confined on simple (one-way) data flow [Dawson2006, Wall2006]. While the coding implementation is simple, finding components that fit into the predefined requirements is not. Not many logic elements are truly independent and are computed on large dataset that overcome the thread dispatching overheads. Thus, the developers need either to use asynchrony-programming models where multiple data copies can be independently manipulated or to restrain fine-grain parallelisms to a small subset of routines. Nevertheless, OpenMP is a perfect tool for fine-grain concurrency but failed on offering enough flexibility for code parallelism on a bigger scope [Gatlin2005a].

6.4.5.1 Applying Synchronous Computations

A particularity of 3DRTS applications is that they generally handle the execution of a large datasets using few instructions. More interesting, this leads to analyze situations where synchronous computations become possible. If GPUs are very efficient to handle vertex and pixel shaders in parallel, CPU-based algorithms could also fork the execution of independent members of a set, such as particle systems or virtual humans' animation update.

6.4.5.2 Managing H-ANIM Human Animation in Parallel

In our system, we can use special virtual humans that we refer as hi-fidelity actors. They represent characters, which have a direct impact on the simulation or scenario. For instance, in one of our projects, we are simulating a crowd within a roman Odeon. To emphasis on specific virtual humans such as senators, we want to offer a highest flexibility by equipped these characters with fully compliant H-ANIM animations. The only downside with this animations system is that the cost to update a single virtual human is prohibitive relatively to more lightweight animations systems. To reduce the performance impact of updating animation skeletons, we separate the rendering and animations update into different threads. This gives the abilities to keep smooth frame rate, where multiple frames can display the same human posture. In addition, we adopt different LODs which decrease both computations and animations quality using similar techniques as the one presented in [Ratcliff2005]. For the animation update, we rely on micro-concurrent routines, which spread virtual humans update on several threads. The Figure 6.7 illustrate that the animation code is sent to several CPUs.

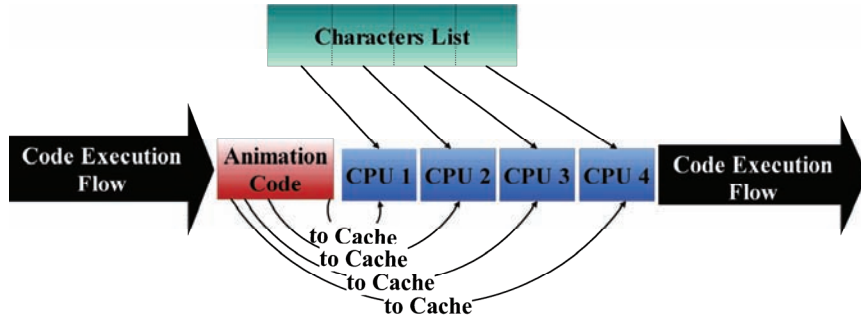


Figure 6.7: Characters animation updates are spread over several threads.

The following listing illustrates the workflow repartition over distinct threads. The implementation relies on [OpenMP].

Listing: H-ANIM Characters Animation Update:

```
void vhdHAGENTServiceBody::performAllAgents(vhtReal rTimeStep)
{
    //prevent concurrent access
    vhdSYNCHRONIZED(*_monitorUpdate);
    unsigned int ui;
    const unsigned int uiSize(vecActiveAgent.size());
    #pragma omp parallel for
    for (ui=0; ui < uiSize; ++ui)
    {
        //required as OpenMP do not allow "parallel for" on vector
        _ActiveAgentVector::iterator it(_vecActiveAgent.begin()+ui);
        it->pAgdata->hagent->setElapsedTime(rTimeStep);
        it->pAgdata->hagent->perform( it->lod);
    }
}
```

6.4.6 Task Scheduling

The creation of independent components running in a thread-friendly and thread-safe environment consist only the first step for processing tasks. Every component will have its own set of requirements from both a memory and processing power perspectives. To conceive a tasks scheduler that can interoperate for combining them into a platform friendly is not a straightforward operation. If running and dispatching tasks among threads is moderately manageable, the opportunity for the scheduler to adopt both run-time profiling information and simulations parameters for optimizing the performance require additional work. Developers need to understand the tasks durations and variations as well as their interoperability. For instance, two elements that need to communicate heavily should not be computed in parallel whenever it is possible to reschedule the tasks execution.

6.5 Cache-Oblivious Algorithms

Today computers architectures describe multiple levels of memory becoming successively slower and larger. Those levels include registers, level-1 cache, level-2 cache, main memory, and disk. The access time increases from one cycle for registers to around 10, 100, and 100,000 cycles for cache, main memory, and disk, respectively (see Figure 6.8). According to the current and future CPUs and memory evolutions [Moore1965], [Hennessy2003], the penalty for algorithms that do not take benefit of this hierarchical memory representation will increase through many caches misses. For addressing this problem, cache-oblivious algorithms were introduced by [Frigo1999]. The idea was to optimize the I/O Models scheme without specific knowledge about the memory block size. They describe that basic problems can be solved using optimal algorithms being cache-oblivious [Frigo1999]. First attempts were dedicated to Matrix and FFT transformation. Later, [Bender2000] gives additional proposals for B-Tree and search tree representation. Cache-oblivious algorithms differ from cache-aware algorithms, as they adapt themselves to any architecture [Chilimbi1999].

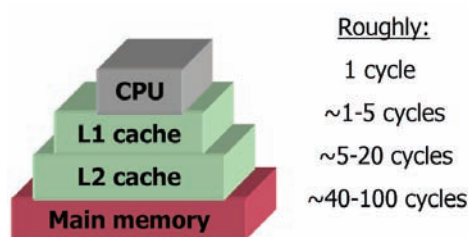


Figure 6.8: Memory multi-level hierarchy.

6.5.1 Computer Memory Architecture

Most algorithms ignore memory architectures. Those driven by small data structures like binary trees suffer huge penalties in accessing their data. Each memory layer works in a similar fashion and is composed of cache blocks. Current architectures use cache lines of ~32-64 bytes. Notable improvements can be achieved by accessing data within the same cache line [Patterson1997, Schulte2002, Hennessy2003]. A cache line's lifetime depends on hardware-specific heuristics [Smith1987]. Decisions are made on a replacement and associative policy within the cache. Sometimes, cache misses are unavoidable [Hill2002]:

- *Compulsory*: A cache miss cannot be avoided; this occurs when some data is access for the first time.
- *Capacity*: The data fit in the cache in previous steps, but due to the renewal policy of the cache, the data was removed from this cache level.
- *Conflict*: Cache trashing due to data mapping to the same cache lines.

6.5.1.1 Data Structure and Cache Coherence

Spatial alignments have an impact on the cache usage. Notably, data structures making intensive use of pointers are not good candidates. Pointers or data that will likely be accessed together should as well be stored together for optimizing memory access [Ericson2003], [Ding2004]. In some circumstances, they may not fulfill the object-oriented programming methodology.

6.5.1.2 Van Emde Boas Layout

The van Emde Boas layout [van Emde Boas1977] (see Figure 6.9) is the standard way of laying out a balanced tree in memory so a root-leaf path can be traversed efficiently in the cache-oblivious model using $O(4 * \log_B(n))$ $B = NbElement / CacheLine$ memory transfers [Agarwal2003].

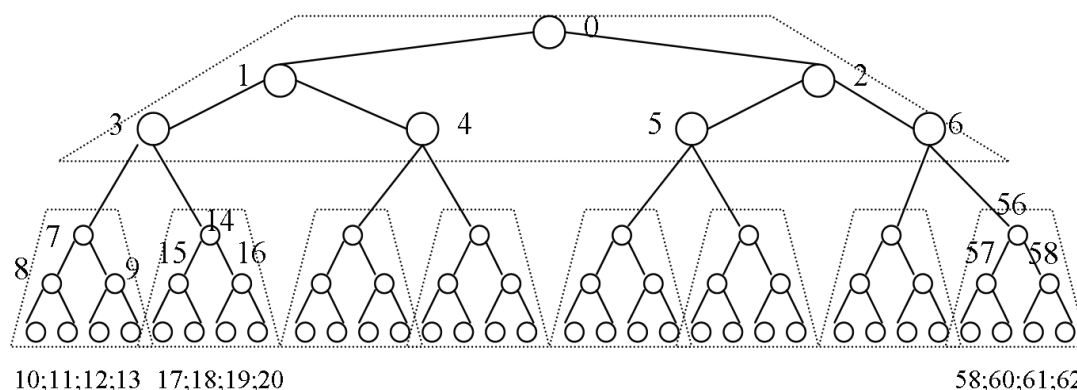


Figure 6.9: Van Emde Boas's tree representation. In this example, each subtree is composed of 7 nodes.

6.6 Cache-Oblivious Hierarchical Tree

To equipped 3DRTS with complex scenes, we need to perform only visible objects in the current view frustum. If scene graph hierarchy offers simple culling methods such as bounding sphere, they do not suffice for massive 3D scenes. In our architecture, we combine multiview and meta-scene graph approaches, which provide multiple accesses on data. However, to speed-up early removal of objects improving the rendering performance, we rely on a cache-oblivious implementation of an Adaptive Binary Tree (ABT). This implementation serves as illustrating that efficient and cache coherent low-level libraries improve the quality of CBD architectures but also offer the opportunity to create more detailed virtual worlds.

6.6.1 ABT-Tree

ABT Trees are very similar to KD Trees [Szécsi2003, Astle2006, Fleisz2006]. At each step, two children may be created based on an axis-aligned splitter. One difference with KD Trees is that the algorithm will minimize the resulting children's AABBs and store all the geometry exclusively in the leaves (see Figure 6.10). Thus, each node becomes a totally enclosed region in space where the internal nodes are used for rejecting the traversal of no visible parts of the environment.

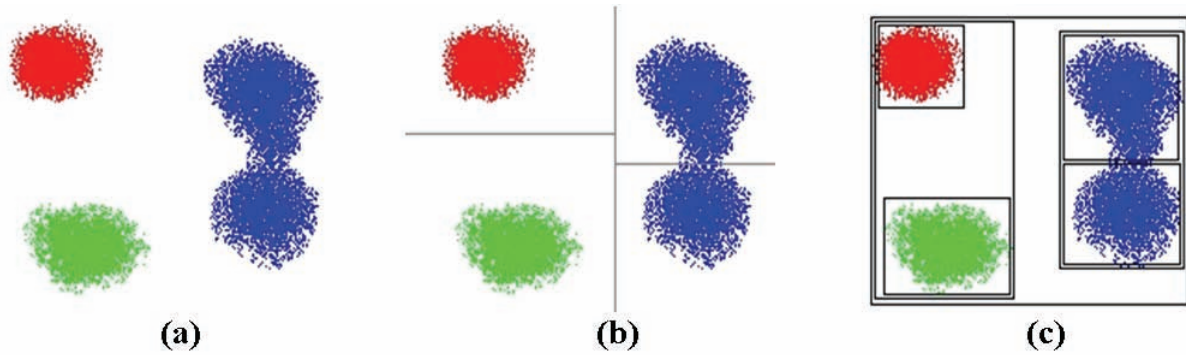


Figure 6.10: a) the original scene b) KD-Tree organization c) ABT-Tree organization.

6.6.1.1 A Ram Implementation

Intuitively, the programmers will provide a RAM implementation where the different node of the tree will be connected by keeping two pointers to the left and right children (see listing below). From a implementation stand-point, this approach is relatively straight-forward, but it also come with many drawbacks for the run-time performance. Since, the storage of all the nodes are not localized into the same slot of memory, this implementation will suffer from many cache misses. During the traversal of the tree, the algorithm will have to follow many pointers which in turn induce cache misses. A cache misses occurs when the data need to be fetch in the main memory rather than in the cache. The penalty to fetch the data result that most of the processing time is spent fetching the data than to traverse the tree. Even worse, a RAM implementation will be directly bound by the memory bandwidth and the replacement of a processor by a faster iteration may result in almost no performance boost. Therefore, to ensure that our architecture can maximize the resource usage, we need to rethink such algorithms. One solution that offers both platform independence and efficient cache usage are cache-oblivious algorithms, like the one describe in the next sections.

Listing ABT-Tree RAM Implementation

```
void struct ABTTree
{
    AABB box;
    ABTTree *pLeftChild;
    ABTTree *pRightChild;
};
```

6.6.1.2 ABT Tree Creation

The creation of such a tree begins with the root node containing a reference to the whole scene AAB. The recursive building method subdivides the local current AAB into two parts along an axis-aligned splitter. Then, each face is assigned to one child depending on their median (see Figure 6.11).

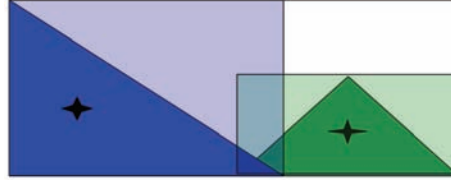


Figure 6.11: Axis aligned bounding box readjustment.

Once the distribution of all faces has occurred, each child recomposes its own AAB containing unique faces on this level. The ending criterion depends on the specific 3D environments and hardware envisaged. The splitting policy will try to minimize the following attributes:

- Space Localization:

$$f_1(n) = \text{Min}(\text{Area}(\text{boxLeft}) + \text{Area}(\text{boxRight})) \quad (6.4)$$

- Tree balancing:

$$f_2(n) = \text{Min}[\Delta(\text{Area}(\text{boxLeft}) - \text{Area}(\text{boxRight}))] \quad (6.5)$$

$$f_3(n) = \text{Min}[\Delta(\sum(\text{ExtendedArea}) - \sum(\text{Area}))] \leq \epsilon \quad (6.6)$$

Epsilon should stay below 5-10% before a noticeable performance penalty. Usually, 90% of the faces will be contained in this 5% extended AAB.

- Scene Complexity:

$$f_4(n) = \text{Min}[\Delta(\sum \text{faces}(\text{boxLeft}) - \sum \text{faces}(\text{boxRight}))] \quad (6.7)$$

$$f_5(n) = \text{Min}[\Delta(\int \text{ressources}(\text{boxLeft})dt - \int \text{ressources}(\text{boxRight})dt)] \quad (6.8)$$

The final equation becomes:

$$f(n) = w_1 * f_1(n) + w_2 * f_2(n) + w_3 * f_3(n) + w_4 * f_4(n) + w_5 * f_5(n) \quad (6.9)$$

The weights will change during the traversal with regard to the engine bottlenecks and scene organization (scenegraph, special effects, etc.). The methodology will also differ for the pre-processing stage and during run-time. At run-time, typically each view will do a full or partial traversal during the different rendering phases (culling, shadow, collision detection, etc.). Thankfully, ABT Tree traversals are really simple and fast to compute. Starting from the last local root, the recursive function tests if either of the two children is in the frustum and if so, continues the tree traversal. When a leaf is reached, all geometry contained can be sent to the next stage in the rendering pipeline. Since each face is unique, no additional tests are needed (like for collision detection). We can also keep a small vertex buffer by leaf and local materials. For reducing the number of vertex buffers used, we can benefit from the neighbor child's location. Therefore, a single vertex buffer could be shared within their limits (generally 65K of 16-bit indices) amongst leaves. This provides a more efficient branching when neighbors need to be proceeding altogether, improving the rendering performance [Wloka2003]. ABT Trees can also be tuned dynamically by simply reordering each moving AAB among their sub-trees. One downside with dynamic trees is that they tend to degenerate with time.

6.6.1.3 Efficiency

All binary trees, notably BSP and KD trees, suffer from depth level. Even with ABT trees' abilities, it remains a significant issue. Assuming the tree implementation is using pointers instead of implicit pointers, we can consider that every time the tree needs to follow a pointer a CPU cache misses will occur. Relative to the

depth level, the number of caches misses will increase accordingly up to a limit where they become more expensive than testing intersections. However, cache-oblivious trees reduce cache misses relative to the cache line size. The traversal becomes then less sensitive to memory access rather than CPU performance.

6.6.1.4 Complexity

ABT Trees, like all binary trees, have a $O(\log n)$ search time [Sedgewick1990]. By using the van Boas cache-oblivious layout representation, the search time can be majored to $O(4 * \log_B(n))$ where $B = \text{CoupleNodesByCacheLines}$ [Fagerberg2002, Brodal2003] (see Figure 6.12).

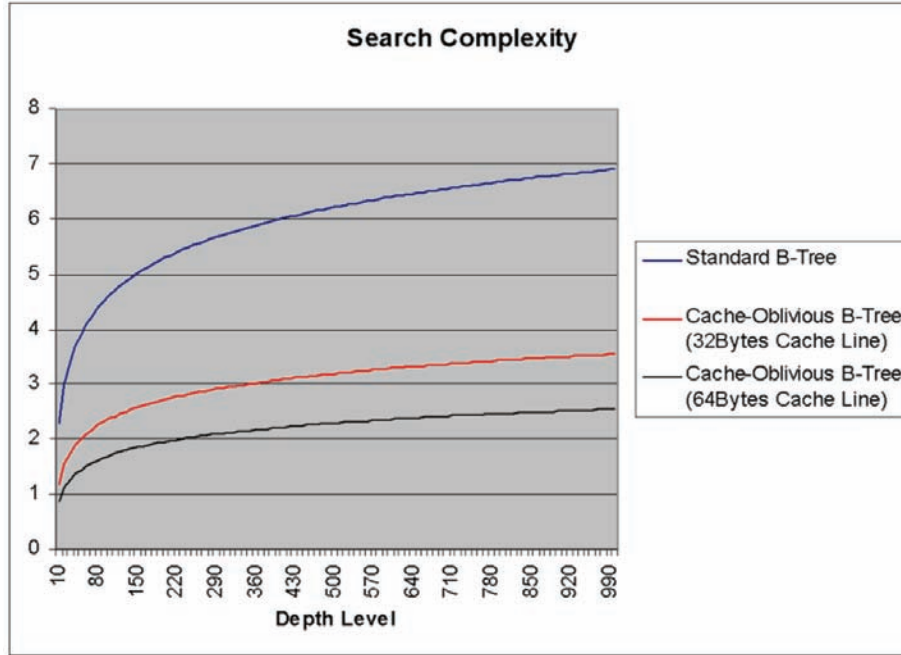


Figure 6.12: Searching time order.

Each node needs to store its local AABB and a pointer (or index) to its two children. Using a naïve implementation, the memory consumption becomes relatively important:

- AABB described by 6 floating-points values: $6 * 4$ Bytes.
- Pointers to its two children: $2 * 4$ Bytes (32-bit CPU) or $2 * 8$ Bytes (64-bit CPU).

Total memory requirement per node is therefore 32 Bytes (32-bit CPU) or 40 Bytes (64-bit CPU).

However, [Gomez2001] have shown that we only need to keep the relative extents for each child, which can be truncated to an 8-bit integer value. This conservative estimate will have a relative error of $1/255$ or $\sim 0.4\%$, which is covered by the average 5-10% AABB overlapping.

6.6.1.5 Exploiting Redundancy

[Gomez2001] have described practical ways for reducing the memory footprint. At each subdivision, 6 extents from the 12 defining the children's AABB comes directly from the parent, as all faces will be propagated to the leaves. For saving a few bytes by node, we store them by couple. Thus, instead of keeping the absolute AABB locally, each couple of nodes will keep the proportion to their direct parent's AABB (see Figure 6.11)

Consequently, 6 bytes are needed to represent the children relative ratio. An additional byte specified through different flags will use this relative position from the two children and reuse the parent value. During run-time traversal, the recursive method will recomputed the local AABB on the fly. Finally, since the last byte

has unused two bits at this stage, we specify whether the left or right children are nodes or leaves (see Figure 6.13). For keeping a cache-oblivious data structure, we store them in 8 bytes, leaving 7 bits unused for the tree itself. For instance, they allow specifying whether the following subtree was loaded, which may be useful for streaming worlds.

63	62	61	60	59	58	57	56	55	48	47-40	39-32	31-24	23-16	15-8	7-1	0
LX	LY	LZ	RX	RY	RZ	LL	RL	LX Ratio	LY Ratio	LZ Ratio	RX Ratio	RY Ratio	RZ Ratio			L?

Since computer cache-line architecture is always a power of two, we aligned our data structure to be 64 Bits or 8 Bytes.

- **L?** Specify if it's a node or a leaf.
- **Ratio L_{x,y,z}**: relative ratio based on the lowest part of the specific axis to enforce that the truncated AABB will be equals or bigger than the absolute one. The ratio is split into 255 elements.
- **Ratio R_{x,y,z}** as above but using the higher part.
- **Flags**: Bit 63-58 specifies which child extend belong to his parent (1 Left Child, 0 Right Child). Bit 57-56 indicates if the left and right children are leaves elements (1 Leaf, 0 Node).

Figure 6.13: ABT tree's node data representation (couple).

Finally, as the subtree will always be a power of two minus one, and as the cache line size is always a power of two, we have 8 bytes available for linking this subtree to the next subtree. As the hierarchy use implicit pointers, 4 bytes are used to provide the index to the first child available. Parts of the 4 additional bytes specify which end-node is connected to a child's subtree. Depending on the cache line size, some bits may remain unused (see Figure 6.14).

N = cache Lines Size (Bytes)					
Byte N	Byte 8	Byte 7	Byte 4	Byte 3	Byte 0
Nodes	C1 ?	C0 ?	1st Children Index	

- **Bytes 0-3**: Index to the first next child sub tree. All sub trees are direct neighbors.
- **Bytes 4-7**: C_x?: Flag specifying if the end-node i is connected to a sub tree.
- **Byte 8-N**: Each group of 8 bytes represents a couple of nodes or a leaf.

Figure 6.14: Cache line organization.

The build routine is done so that all subtrees coming from the last subtree are direct neighbors (see Figure 6.15).

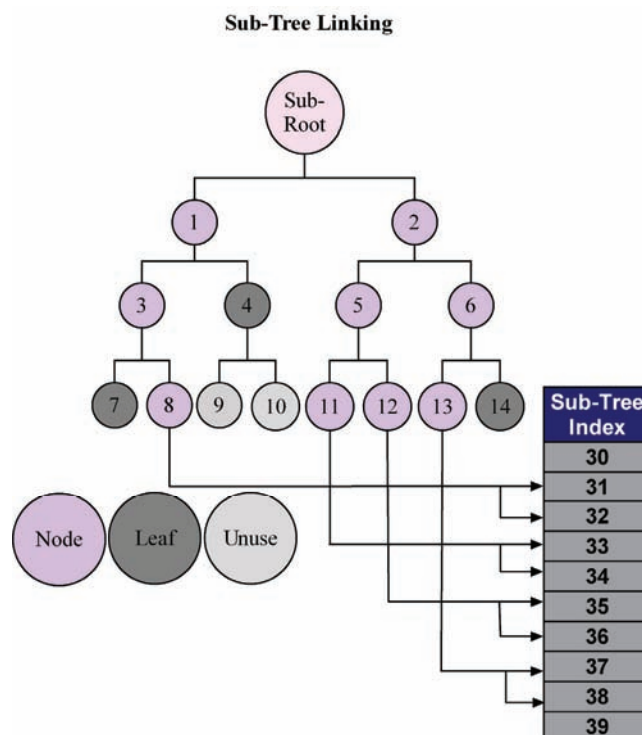
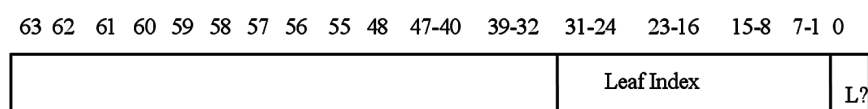


Figure 6.15: Subtree organization with linking to following subtrees.

Now consider a leaf, whose composition is made of 8 bytes. 1 bit is used to specify its condition. Then again, it may depend of the number of nodes the system may use. On 32-bit systems, 31 bits are generally enough; leaving 4 bytes that can be used for additional information (mainly for dynamic management of resources) (see Figure 6.16). However, 64-bit programs may want to use more nodes and therefore take more than 31 bits.



- L? Specify if it's a node or a leaf.
- In this configuration, 31 bits are used for referencing the leaf, the last 4 bytes aren't used for the tree itself, but can store extra information (like for streaming worlds).

Figure 6.16: ABT Tree's leaf data representation.

6.6.1.6 Performance

For analyzing the performance of this approach, three different implementations were used (see Figure 6.17):

- *Intuitive*: 6 * 4 bytes for storing the AABB and 8 bytes for the children's pointers (32-bit CPU), or 64 bytes for a couple.
- *Using redundancy*, 8 bytes for the AABB, 8 bytes for children's pointers, or 48 bytes per couple
- *Cache-Oblivious* (64 bytes cache lines): 8 bytes per couple + 8/7 extra bytes needed for implicit pointers, giving an average of 9.14 bytes. The global memory requirement depends on the tree balancing (see Figure 6.15).

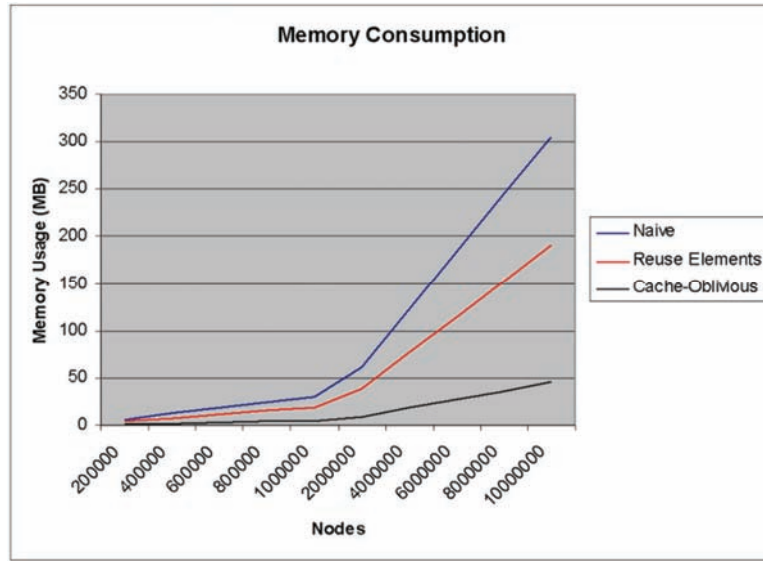


Figure 6.17: Memory usage across implementations. The memory consumption for the cache-oblivious depends on the tree's balancing.

With regard to the 8-bit integer value for representing the children's extents, the overhead to convert them back were measured. Experiments made on 16k randomly distributed leaves have shown an overhead of about 7%, which is clearly compensated by the better cache-friendly design.

6.6.1.7 Results

The Figure 6.18 illustrates some test-units applications dedicated to analyze and validate the run-time performance of our ABT Tree. In Figure 6.18A, we generate an area of $10'000^2$ graphic objects that stress our geometry culling algorithm implementation. In Figure 6.18B depict a test-unit featuring two distinct viewpoints. The top view represents the final 3D scene, while the bottom view provides diagnostics information. In Figure 6.18C, we display the Axis Aligned Bounding Boxes for every node in our hierarchy.

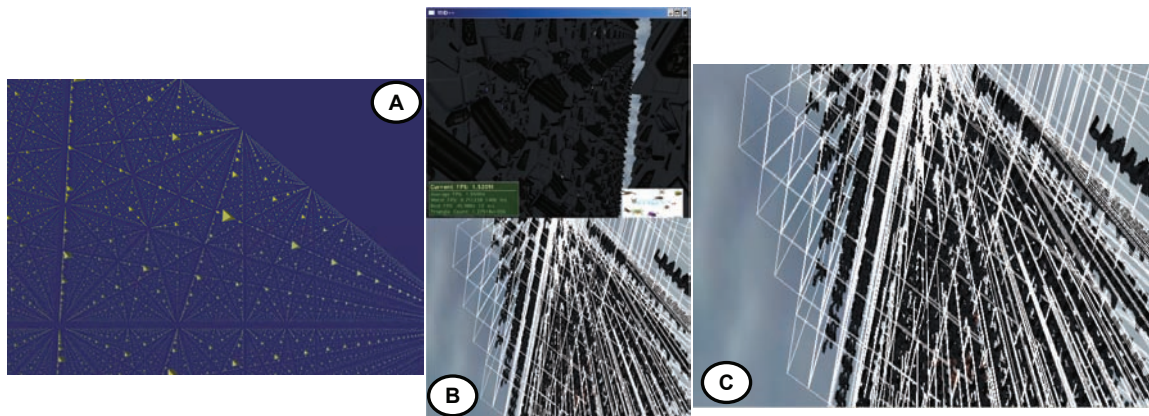


Figure 6.18: Testbed applications for analyzing and validating our ABT tree implementation.

6.7 Low-Level Libraries Optimizations

In the course of our developments, we dedicated attention to develop efficient low-level libraries, which are intensively used in our simulations. Notably, we developed a math library relying on the SSE/SSE2 instructions set [Devir2002, van Leitner2003] and that improve float points operations [Shewchuk1997, Cormen2001, King2001, Green2003, Lengyel2003, Tremblay2004, Ericson-NRGC2005, Lomont2006]. The

following listing illustrates how SIMD may decrease the number of CPU cycles. The average speed-up using our SSE implementation is around 50% faster than the default SISD implementation.

Listing SSE instructions:

```
inline vhdVector4 vhdMatrix4::operator*(const vhdVector4 &vec) const
{
    F32vec4 result;
    result = (F32vec4) _mm_shuffle_ps(vec.m128, vec.m128, 0x00) * _L1;
    result += (F32vec4) _mm_shuffle_ps(vec.m128, vec.m128, 0x55) * _L2;
    result += (F32vec4) _mm_shuffle_ps(vec.m128, vec.m128, 0xAA) * _L3;
    result += (F32vec4) _mm_shuffle_ps(vec.m128, vec.m128, 0xFF) * _L4;
    return vhdVector4(result);
}

inline vhdMatrix4 vhdMatrix4::operator+ (const vhdMatrix4 &B) const
{
    vhdMatrix4 res;
    res._L1 = _mm_add_ps(_L1, B._L1);
    res._L2 = _mm_add_ps(_L2, B._L2);
    res._L3 = _mm_add_ps(_L3, B._L3);
    res._L4 = _mm_add_ps(_L4, B._L4);
    return res;
}
```

In addition, time critical code section can be optimized by taking advantages of intrinsic functions for a better control on the cache memory [Wall2"5, 2006]. These offers fetching data into memory cache lines or to write directly data to memory without accessing the different cache layers hierarchy. Moreover, C++ compilers may offer specific optimizations using #pragma [Intel_ICL2005].

Listing C/C++ optimizations: cache control and loop unrolling

```
//fetch data with "non-temporal" tag
_mm_prefetch( (char*) _ptr, __MM_HINT_NTA);

//store directly to memory, not to cache
_mm_prefetch( _ptr, __MM_HINT_HTA);

//loop unrolling
#pragma unroll(4)
for (i=1; i<m; i++)
{
    d[i] = c[i]+1;
}
```

Finally, compilers can parallelize code sections when programmers use keywords such as *restrict* and *const* that serves as hints during the code generation. This may reduce aliasing issues. Aliasing occurs when multiple references to the same storage location exist [Ericson2003]. As compilers must remain conservative, they cannot perform the optimization. The following listing describes some low-level optimizations using the concept explain herein.

Listing Aliasing:

```
//stores may alias loads, so must perform operations sequentially
void vecAdd(float *a, float *b, float *c)
{
    for (int i(0); i<4;i++)
        a[i] = b[i] + c[i];
}

//independant loads and stores. Operations can be performed in parallel
void vecAdd(int * restrict a, int *b, int *c)
{
    for (int i(0); i<4;i++)
        a[i] = b[i] + c[i];
}
```

6.8 Multi Layer Concurrent Programming Techniques

Designing software architectures for concurrent 3DRTS applications do not entirely rely on the dedicated hardware. Whatever choices hardware manufacturers agreed for their systems; the software can adopt a unified approach to this problematic. The software architecture has to adopt ideas for decoupling the code to determine which portions can or cannot run simultaneously. Such systems scale more easily within high multitasking environments if they can be separate into independent and parallelizable sections. The paradigm shift involve to move from a sequential single-threaded architecture to a concurrent model is a complex process. The design process has to provide the interdependences between components and determine the best way they can run in isolate environments. This design work has to be done before entering into the concrete coding implementation. This critical element force to analyze several factors to understand how the system will reacts. The consequences are twofold: the way the code will be computed and how the underlined hardware processes it. Figure 6.19 illustrate the main aspect-graph that provides concurrent access synchronization on shared data.

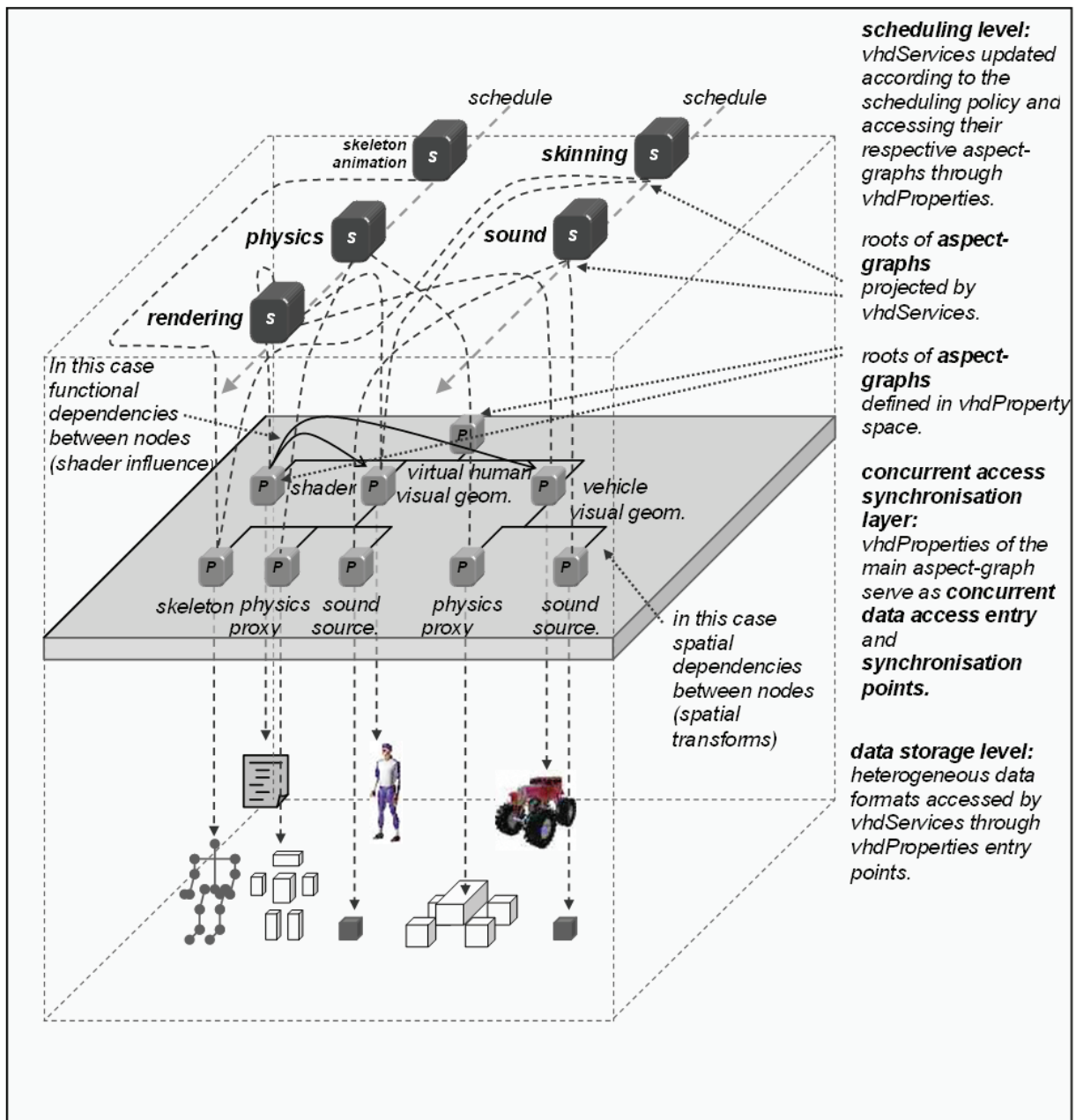


Figure 6.19: Main aspect-graph composed of vhdProperty serving as a concurrent access synchronization layer to access data underneath.

Concurrent programming presents different layers of parallelization. The parallelization can be applied on the instructions level taking full advantages of hardware specific functionalities or from components perspectives, where independent module can run isolate tasks simultaneously. The literature refers these two concurrent models as fine-grain and farm parallelism. The hardware architecture defines how the fine-grain or farm parallelism can be applied. For instance, systems architectures that featured custom memory for each hardware elements (sound memory, graphics memory...) or that features many intrinsic functions will become good candidates for fine-grain parallelisms design. In the other hand, systems architectures relying on uniform memory or which feature symmetrical cores allow efficient farm parallelism design. Naturally, most current hardware architectures are a combination of these two approaches, which indicate that a good balance for both software methodologies has to be considered. In the general purpose, it is better to confined fine-grain parallelism operations on low-level libraries that are intend to be intensively used. Such candidates include math or physics libraries. The farm parallelism is only possible if the software design isolate significant portions of code. CBD enforce those requirements but it will not free developers to profile the code for discovering typical concurrent bottlenecks such as data synchronizations or resources locking. In any cases, the software will have a primary thread that will act as a system controller. It will be responsible to dispatch the work and spawn threads among the available processors. Then depending on the internal methodology that a package is using, the primary thread will have the responsibility to determine which processor will be allocated. In practice, it is unfortunately very difficult to implement as it rely more on fine-tuning workflow balancing than from a conceptual methodology. Moreover, it is not always possible to send code to dedicated processors due to OS restrictions.

6.9 Concurrent System and Memory Management

Different policies for managing memory within a concurrent system exist [Berger2000]. In our system, we rely on separating memory blocks into three different categories for each thread (see Figure 6.20). The first category represents memory blocks that do not contain any valid information. If in single-threaded applications, this would be related to not initialized or deprecated objects. Here, theses memory zones can contain valid private information from another thread. In other words, the thread is denied to access those zones. One might argue about the reason of denying memory zones that contain valid information. In fact, by keeping some memory zones private to a dedicated thread, provide the ability to remove synchronization primitives on data structure accessed exclusively by a single thread. The second category represents the private memory dedicated for a thread. The advantages are twofold: this reduces the overhead of thread-safety locks, by granting a free access on single-threaded data and this allows creating memory zones localized with the hardware thread. Depending on the hardware architecture, a hardware thread may have a dedicated memory channel that can be used directly [Hoeftler2005]. Finally, the last category covers memory zones that are shared between threads. Those memory zones required to provide synchronization routines on top of the data structure, which can be done by forcing accessing data through assessors. In fact, shared memory zones should forbid the usage of providing direct pointers on those data as well as trying to minimize their size, which is critical for performance.

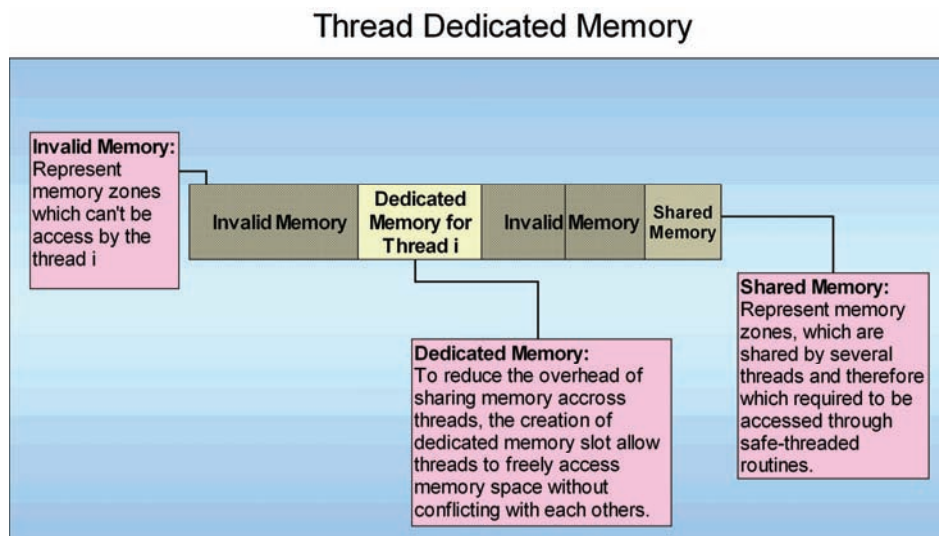


Figure 6.20: Every thread see the repartition of memory zones into three categories: invalid memory, dedicated memory and shared memory.

6.10 Tasks Dispatching

Typical 3DRTS require the usage of several components such as 3D renderer, sound, AI, physics, scripting... When it comes to configure the system for dispatching the workflow over the different hardware threads, different situations need to be analyzed. Primarily, we need to profile and collect information about each task. At this level, a task represents a component's update. The data collected identifies the task, the CPU on which it run and the duration. The Figure 6.21 describes a scenario where four threads are used. In this model, one thread is dedicated for managing the system and another one for the rendering. The last two threads are responsible to update the remaining components. Here, the idea is to let the system run and collect statistics. This information allows rescheduling tasks. This is done using three levels of priority. The highest priority goes to tasks that need updating at high frequencies, such as 3D rendering or physics. The remaining tasks will then be allocated to a CPU, assuring that every task have a chance to run at an acceptable frequency. Finally, the internal tasks update ordering will be altered for minimizing the cross dependencies in-between tasks running on separate threads. This may avoid that one processor stall due to the internal tasks ordering.

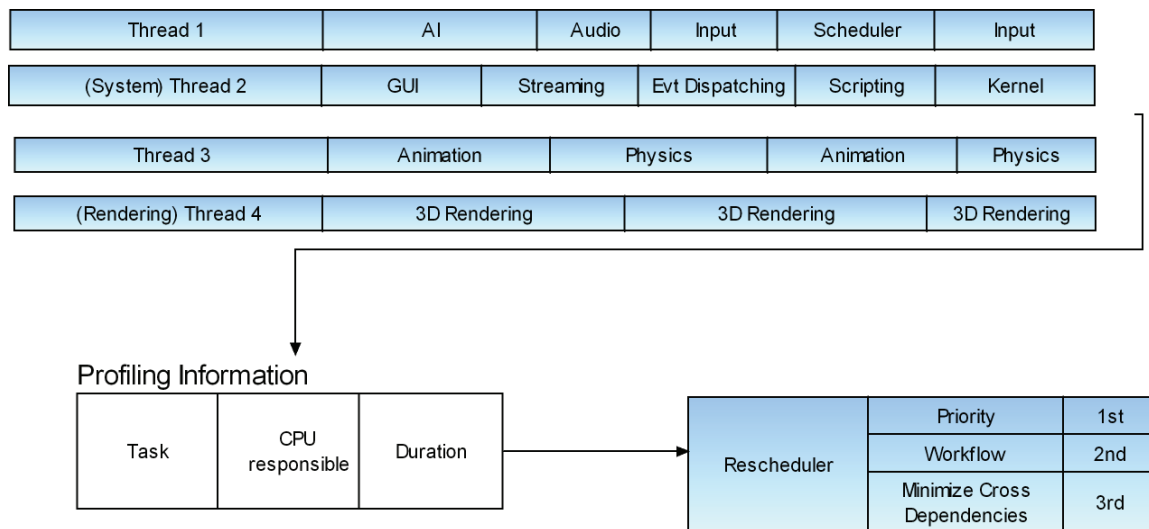


Figure 6.21: The component's update execution is spread over different threads (here 4).

The creation and repartition of the different task scheduler-handling component's update are configurable through XML tags. The next listing illustrates the XML code corresponding to the Figure 6.21.

Listing XML for scheduling component's update:

```
<schedule>
  <serviceUpdate className="vhdAIService" name="" />
  <serviceUpdate className="vhdSoundService" name="" />
  <serviceUpdate className="vhdInputService" name="" />
  <serviceUpdate className="vhdSchedulerService" name="" />
</schedule>
<schedule>
  <otherServiceUpdates />
  <serviceUpdate className="vhdPythonService" name="" />
  <serviceUpdate className="vhdLuaService" name="" />
</schedule>
<schedule>
  <serviceUpdate className="vhdHAGENTService" name="" />
  <serviceUpdate className="vhdPhysicsService" name="" />
</schedule>
<schedule>
  <serviceUpdate className=" vhdOSGViewerService " name="" />
</schedule>
```

6.11 Data Synchronization

One major difficulty with concurrent system is to keep data consistent by ensuring that multiple components accessing the same chunk of information will get up-to-date status. Different approaches exist [Rajwar2002]. One solution consists on keeping only one single copy of every data. In this situation, every time a module wants to access data, the system architecture ensures the content is up-to-date (since it is unique). However, this increase the need of mechanisms that prevents undesirable simultaneous access, such as one module reading data simultaneously than another writing new content. This is extremely important as otherwise, corrupt information will occurs. The highly concurrent approaches of new computers hardware prevent such approaches as each core generally keep its own cache and processors will stall, waiting on mutex and other synchronizations routines. Thus, in our approach, we rely on duplicating the data into each module, relying on the system kernel to propagate the modifications as describe in Figure 6.22. One might argue that such approach increase the memory footprint and force the system to duplicate and propagate the same information but they are many advantages of applying such methodology. First, the multi-layers aspect of CBD, promotes code reuse by taking advantages of low-level libraries that will be independent of each other. They will already keep internally a duplicate copy of basic information such as position and orientation in 3D space. Moreover, as the system will run on multiple threads, reducing the level of data synchronization, will improve performance even at a cost of additional memory footprint. In effect, modern CPUs are not fully used due to memory latency. In addition, in many situations, when multiple modules need to share the same data, only few (and generally only one) modules will modify the data, while the other modules will only read the information. One limitation with this approach is that it may lead to race condition if two components act as writers. Hopefully, this situation is uncommon. In the example taken from Figure 6.22, we can see that the AI engine is changing the position and orientation of an agent, while the 3D renderer component only use the position and orientation for culling and rendering purpose. However, the 3D renderer does not affect them directly. In our system, the kernel is responsible to propagate these modifications. Here, both the AI and 3D renderer modules run in parallel with separate frequencies. We want to prevent that one module has to wait for the completion of another module due to data synchronization. So instead, of applying the changes on ISOs immediately, the system kernel will apply these modifications just before the next update of the client component.

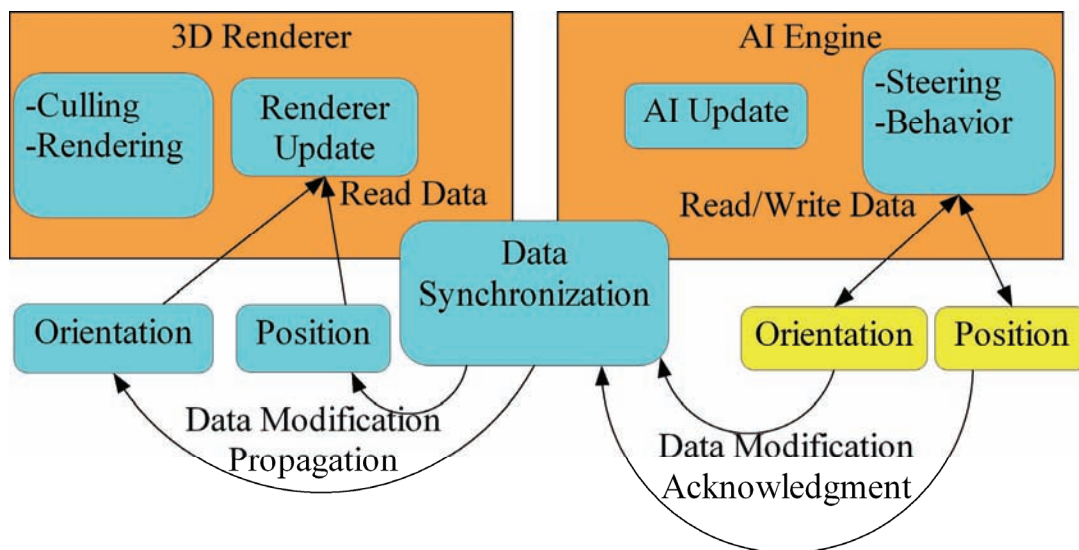


Figure 6.22: Data synchronization pipeline between components.

Let us take the following example; imagine the AI engine runs at 30FPS, while the 3D renderer runs at a solid 60FPS. At some point in the simulation, the AI engine wants to modify the current position and orientation of an agent. If we would apply the modifications directly and in the worst case, the AI thread will have to wait for the completion of the 3D renderer update or around 16ms. This would greatly affect AI component frequency and overall system performance. Not only the component update will become slower and inconsistent, it will also depend on the companion component update. In effect, if the 3D renderer run only at 30FPS, the AI thread may wait (again in the worst situation) 33ms, which reduce the performance by two, where 50% of the AI thread is spent, waiting on the 3D renderer update. The Figure 6.23 illustrates the complexity of proper tasks ordering minimizing the inter-threads communications latency.

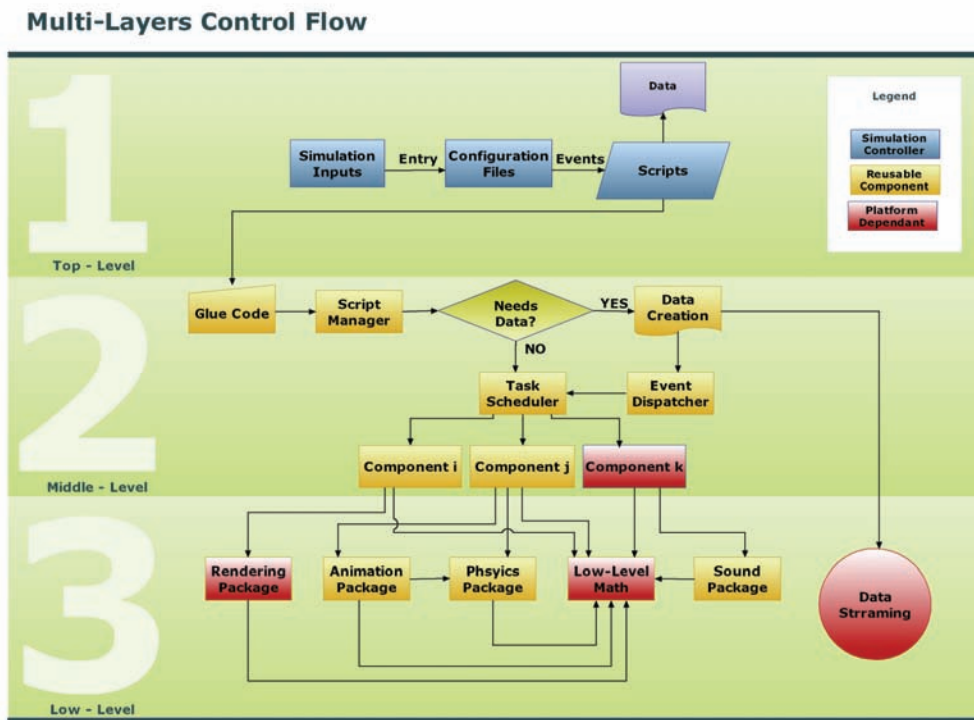


Figure 6.24: The main three control flow layers.

6.13 Managing Processing Throughput with Concurrent Systems

Managing the good repartition of workflow between CPUs is difficult and requires an interactive process. Many parameters enter the equation, which include resources usage and inter-threads communications bottlenecks. Some methodologies can be proposed to analyze the system performance and how reordering components may increase the performance [Choi2000, Schultz2005]. The Figure 6.25 illustrates the different steps helping in this process. Firstly, the purpose of the processing throughput management is to reschedule the tasks for better efficiency. This may come from semi-automatic techniques that include to provide hints to the systems through configuration files or to take advantages of statistics provided by the internal profiler. Running the same application on several platforms help to understand where are the performance limitations such as GPU or CPU bound situations. Another important fact is that the content may not be very efficient due to specific run-time constraints. In many circumstances, artists will create content that will not perform efficiently by creating too complex geometry or requiring heavy pixel shaders' resources [Eiße2006]. For analyzing the component workflow, the usage of profiling tools such as VTune help to analyze the resource usages may even help to discover suboptimum or inefficient code. The solution to improve systems performance may be separated into three different categories. The first category covers the tasks delegation. It consists to observe the effects of changing the repartition of components update on different cores or by modifying their internal ordering reduce the inter-threads communications latency. Often, the serial code that cannot be executed in parallel become the principal bottleneck. The second category isolate component one by one, as sometime as bad implementation of a single component can reduce the hope of acceptable performance. In our experiments, some of the earliest implementations of our H-ANIM component was not efficient and whatever combination of components used, the system performance was low. By replacing the H-ANIM component with a better implementation, the overall system performance was greatly improved. The third category concerns the side effects of multi-tasking systems with the problematic of race conditions and cache coherence. Cache management is very important on lower level libraries, which have a limited code base but run on large datasets. On higher level, OOP and CBD increase cache misses due to indirections through inheritance and pointers. In addition, it is important to runs the systems with a number of active threads corresponding to the number hardware threads available to reduce unnecessary context threads transitions penalty. Therefore, being able to run the same system into different configurations modes allow better fine-tuning. Finally, special attention related to other bottlenecks such as I/O operations need to be investigated seriously.

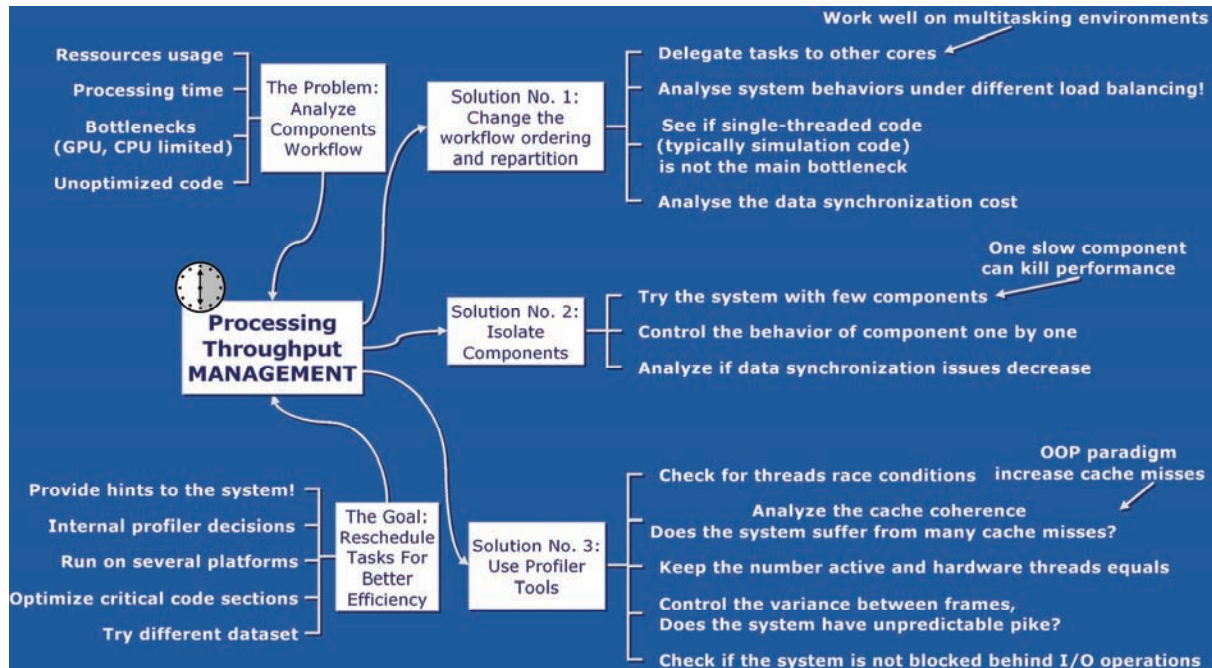


Figure 6.25: Processing throughput management for concurrent systems.

The important numbers of parameters that affect system performance in a concurrent world prevent the creation of generic approach that would help to increase performance immediately. Only strong methodologies and iterative processes that analyze and correct the system components exist. This requires patience especially that the configuration may work only on some platforms or with some datasets [Rabin2004]. This requires multiple configurations for each situation. Even so, the processing throughput management can be applied in the early stages of development, this is only in the later stage that the overall configuration and tasks dispatching can be made in an efficient way, in regards to the considered platforms and datasets.

6.14 Conclusion

Mission-critical software will always be highly dependent on processing throughput. Therefore, systems architects need to provide solutions that combine high performance and good flexibility. Our architecture clearly benefit from modern hardware capabilities such as advanced GPU chipsets and multiprocessors CPUs. We introduced efficient low-level algorithms developed for situations where performance matters. This gives the opportunity to endorse high-level architectures styles with a limited performance impact. Notably, we illustrated techniques for system management in a concurrent world. Our approach deal with those issues by providing mechanisms helping in the development of concurrent software together with configuration modes allowing to adapt the component's update executing on distinct schedulers. The trade-off coming with high-level representations is worth due to better flexibility and readability. In effect, 3DRTS simulations follow the old 90/10 rule, which indicate that you should only focus on run-time performance in the 10%, keeping high-level representations in the 90% of the code where expressiveness and fast coding are more important. At the same time, our design keep in mind the needs and requirements for non-programmers. The goal is to free them of low levels implementation details and specific hardware constraints, whenever alternatives solutions exist. This is particularly important, as current systems architectures are not anymore confined only for programmers. This chapter closes the section on the intrinsic functionalities of our design. The next chapters will introduce data-driven mechanisms equipping our framework with interpreted scripting languages that unleash multitasking operations for handling multi-agents behaviors and simulation events.

Chapter 7

Scripting

Scripting languages increase the system flexibility. They lead directly to early prototyping and dynamic content creation. Scripting languages reduce the learning curves and free developers from many technical issues allowing non-programmers to interact with the system [Poiker2002].

7.1 History

The dramatic growths in complexity that affect 3DRTS software oblige developers to move from low-level libraries to more high-level representations. In effect, only small subsets of the entire code base are time critical elements. Significant amount of code can be deployed with more high-level, better and cleaner representations at a minimal performance costs [White2001]. Thus, the appearances of scripting languages use at run-time with mission-critical software are becoming widely accepted. Scripting languages such as [Python], [Lua], [Perl], and UnrealScript [Sweeney1998] run within a virtual machine. These virtual machines can be embedded within existing systems. They execute scripts expresses in platform-neutral Byte code instead of the processor-specific machine code generated by languages such as C and C++. Using such architectures allows hiding many complex details such as memory managements and OS functionalities. Isolating these details also limits the severity of errors that can be caused by programming bugs. In effect, scripting languages employ weak data typing, which means that variables can contain different types of data at different times, and automatic type conversions that allow one data type, such as a number, to be easily converted to another, such as a string. This allows a great deal of flexibility in developing algorithms. In addition, scripting languages provide high-level objects such as strings, lists, dictionaries, and the like, which improve productivity [Ousterhout1998].

7.2 Interpreted vs. Compiled Languages

Programming languages generally fall into one of two categories: *compiled* or *interpreted*. They are several differences between these two categories. With a compiled language, the code is reduced to a set of machine-specific instructions before being saved as an executable file using a compiler (translator that generates machine code from source code). With an interpreted language, the code is saved in the same format than entered (systematic executor of source code, where no translation takes place). Many people start programming using some form of interpreted code. Typically, interpreted languages are easier to understand than compiled languages, though they may offer less flexibility or functionalities. Only in recent years have interpreted languages been used for mission-critical software, especially due to their relative performance overheads. This can be easily explained as a program written in an interpreted language is read by another program (usually written in a lower-level language like C), called an interpreter. While interpreted scripting languages do save time and efforts, the tradeoff is that the code execution tends to be slower [Bagley2005]. Moreover, the interpreted languages are generally less powerful than compiled languages as they have restricted access to low-level functionalities, like controlling devices directly, addressing memory, etc. Modern systems architectures for mission-critical software combine both compiled languages and interpreted scripting languages for better flexibility and non time-critical code, built on top of compiled core functionalities that enhanced the control on the underlined OS and hardware.

7.3 Related Work

In different fields of software engineering, systems architectures are well described within the literature; however, this is uncommon for 3DRTS systems. Fortunately, this is beginning to change. [Rucker2002, Gold2004] and [McShaffry2005] try to teach game development using reusable architecture. They describe how design patterns can be applied to 3DRTS, and discuss game systems by their architectures. They provide fundamental information on software practices, especially those used in complex architectures from other fields [Lakos1996]. Many problems associated with large-scale software development have been explored.

[Bass2003] provide valuable information that can be applied to generic systems. [Parise2005] presents scripting systems overview comparing two approaches: to embed an existing language or to develop a custom alternative. He describes how scripting system can ease the management and persistence of ISOs as well as related requirement such as editor and diagnostics and debugging capabilities. [Kalogirou2005] presents a scripting engine relying on Stackless Python [Tismer2000]. He proposes that each actor should run in its own context, by dedicating one thread for every actor, similarly than an OS process does. He pointed also that multithreaded environments are difficult to handle properly, especially for inexperienced programmers. His approach consists of benefiting from Stackless Python and tasklets, which can solve the scheduling overhead, memory costs and slow thread creation. This is achieving by non-preemptive multithreading routines. This solves synchronization problems. A non-preemptive environment rarely needs locks and can be deterministic. In effect, in a non-preemptive environment, a semaphore is just a variable, and there is no need to lock between reading a variable and updating it. Over the year, different attempts for addressing these higher levels objectives have been developed [Varanese2002, Garcés2006]:

- Custom grammars or languages that are developed with a specific purpose in mind.
- Embedded scripting languages like [Python] or [Lua]. Here, there is good support from the community but the languages may be too heavy for embedded platforms.
- CORBA: Allows communication across processes, but its overhead may limit its usage [Ciger2005].

Ideally, when designing virtual worlds populated by individual entities, we would expect that every agent react independently. Thus, the most intuitive approach is to dedicate a separate flow of control for each character. From a programming perspective, we can envision allocating a dedicated thread to every single agent. However, the overhead of thread context switches and memory latency makes this approach prohibitive. For optimal performance, the number of heavyweight threads should match the number of cores available. Moreover, developing and debugging within a multithreaded environment is much more difficult than within a single threaded environment. Only a few team members may have the skills and knowledge required to develop multithreaded component-based systems. Therefore, a key objective is to provide a software architecture that can be manipulated through higher-level of abstractions while keeping real-time performance and a multitasking representation [Ramsey2005].

7.4 The Python Interpreter and Component-Based Software

CBD provides a layered architecture style that divides the system functionalities into a hierarchy where each layer connects to the layers above and below it [Rene2005]. This approach promotes reusability by keeping the application specific code at the topmost levels, and allowing developers to reuse the lower layers [Duffy2004]. The bottom layers are low-level libraries that are optimized to run on specific platforms, while the upper layers are used for communication with the system. Here, an independent plug-in can communicate with any top-level services. This allows exposing engine functionalities to scripting interfaces with ease. In this architecture, the Python interpreter handles the connection between C++ objects and Python objects by reflecting them in a safer environment. This frees the developers from common C/C++ issues like memory management or type checking, increasing the development time that is available to experiment with ideas. The following sections will describe how a standard interpreter can be extended with more advanced features toward meeting the specific requirements of 3DRTS systems.

7.4.1 Benefits of using a Scripting Language

Many programmers especially those involve with mission-critical applications think their design and integration only through ideas and language intrinsic functions. The leads to reduce the scope of approaches that can be apply to resolve one problematic. Today, C++ is the principal language used for 3DRTS. Its run-time performance [Stroustrup1994, Bruckschlegel2005] remains its principal advantage. However, it may not be the best choice for all tasks, as the C++ programmer is burdened with many restrictions and inconveniences like memory management and compilation time. This reduces the disposal time available to experiment with ideas. Scripting languages allow rapid development of behaviors and simulation events without the pitfalls that await the unwary C++ programmer. They also reduce the C++ code dependencies and C++ subclasses level by clearly separating the logic written in C/C++ to the data manipulation controlled by scripts. By developing with different programming languages, programmers can expend their knowledge and help to resolve problems

outside the language constraints. Naturally, being able to understand the limits and potential of the different languages is the key for successful and efficient software development, but the main design should not be dictated by the languages. Even so, programmers can be tempted to use languages weaknesses for specific purpose; it should not be enforced, if the result goes against the main design principles. Good approaches consist to think about the goals to achieve and then to find the most appropriate ways to accomplish them, which may not be restrained to one programming language. Many times, combining multiple languages may offer the best environments. Some programming languages perform some tasks better, where the valuation is not always the processing time to execute it, but may depends on other factors such as flexibility or fast developments. Thus slower languages, such as interpreted scripting languages may perform the job more accordingly [Sweeney2006]. Usually, C++ code will run faster, but in today's applications, there is an important amount of code where performance is not an issue [White2001]. By developing this code in a scripting language, it is possible to achieve a much more efficient development process. For example, one interesting feature over the static nature of C++ is that it is possible to add new member variables at run time based on the needs, avoiding creating objects which contain all imaginable states at all times. The global vision may also induct to create functionalities that the targeted programming language does not support. For instance, if the language does not support hash map, they can be integrated using custom routines. This is important when working with new design approaches and for which, programming primitives do not exist. This is particularly the case with concurrent models. Nowadays, there are not many languages, which were developed with a good emphasis on concurrent programming models. 3DRTS programming language of predilection remains C++, which do not provide good primitives for working within a multithreading environment. Moreover, the technical approach of thinking in a concurrent world is not a straightforward task. Thus, being able to manipulate the system within a thread-safe sandbox is important. As the current set of programming language, still lack the set of primitives to provide, build natively such systems. The difficulty is to extend the programming potential with new mechanisms while keeping the system managed by clients' programmers. Programming conventions clearly help to provide customized features that can be used within project as showcase by the results from [de Sevin2005] (see 10.5).

7.4.2 Disadvantages of Scripting Languages

Using multiple-languages for developments adds an extra layer of complexity. It may be harder to debug in both languages simultaneously or to port the languages to different platforms, and time must be spent on maintaining the glue code that ties the languages together. This is particularly true with dynamic languages like Python [Beazley1999], which suffer from a different set of run-time problems than a compiled language. Fortunately, those issues are easier to manage than their C++ counterparts.

7.4.3 A Proper Usage

One problematic that surface with integrating an interpreted language is to clearly specify which elements can be developed within scripting languages or should remain as C/C++ modules [Berger2002]. Once the scripting language is running, many elements of the simulations can take directly advantages of it. The idea is to promote data-driven systems by allowing designers to describe simulations events such as playing animations or executing behavioral scripts. However, as this higher-level offer to interact with many components, it may end-up controlling too many elements resulting in an incremental increase of complexity. In effect, creative people or simulations designers are likely to explore new ideas. Often, they may try to reach the application limits by overloading the system by intensive processing scripts [Tozour2002]. [Jaffe2005] described two categories separating situations where to use and not use scripts. If scripts should not be used for developing algorithms that require complicated data structure or that rely on heavy floating-point operations, the use of scripts is beneficial for the following situations:

- Event handling for both UI and world systems.
- Basic operations that are already implemented by single function programs.
- Heavy text-based processing.
- Fast prototyping.
- Data types rely on list or hash map.

7.4.4 The Choice of Python

From the existing interpreted scripting languages, both Python and Lua are extensively use in 3DRTS production. Lua is smaller, probably easier to embed in an application, and has some nice language constructs [Ierusalimschy1996, Harmon2005, Schuytema2005]. Both languages provide similar functionalities and either language is a safe choice that increases the productivity over single language systems [Gutschmidt2003]. For the system described herein, we chose Python because it has more extension modules than Lua, more books written about it and is generally considered being easier to learn [Dawson2002, Riley2003a]. This is particularly important for introducing the system to newcomers who may have limited experience with writing code. Using an existing scripting language avoids the typical limitations of custom languages (lack of documentation, limited functionalities...). Python is an excellent choice for game systems, because it is powerful and can be easily embedded and seamlessly extended with C/C++. Additionally, the literature and development tools provide great opportunities for benefiting from the work of others.

7.4.5 Code Bindings

Binding C++ modules with Python can be difficult to maintain if done by hand. Thus, a system for generating the glue code way is essential. Different tools helping to automate this task are available [CXX, Python, SIP2005]. Another alternative is libraries like [Bilas2001] or [SWIG], as generic system of making C++ functions and classes available to a scripting language. Most of these glue code systems work by parsing a C++ header file. Thus, they are restricted to what a C++ header can expose. The implementation described herein relies on [SWIG], which provides automatic glue code generation with little extra work. Every C++ component exposed to Python will generate a Python module. The listing describes a SWIG interface file used to expose C++ classes to Python. This interface consists of a list of headers from which the glue code will be created. SWIG also handles special situations such as templates, method overloading, and method hiding.

A Plug-In SWIG Interface file:

```
/* File : pyvhdPathPlannerService.i */
%module pyvhdPathPlannerService

%{
//the code will be inserted in the generated file as it is
#include <vhdPythonInterpreter/vhdPythonInterpreterSWIGDefine.h>
#include <vhdFundamental/vhtBasic.h>
#include <vhdPathPlannerService/vhdPathPlannerServiceBody.h>
}%

//as Python do not handle template, explicit instance used within
//the API needs to be explicitly instantiated
#include std_vector.i
namespace std {
    template<STLVectorFloat> vector<float>;
    %template(STLVectorVector3) vector<vhdVector3>;
}

//this method will not be exposed through Python
%ignore vhdPathPlannerServiceBody::removeAllNodes;

//macro or types may not be always handle properly
#define __int64 long
#undefendef

//ask SWIG to expose the class and its public methods
#include <vhdPathPlannerService/vhdPathPlannerServiceBody.h>
```

Moreover, as we want to operate with existing C++ objects, the interpreter is built with the functionality to generate a Python reference object bound to its C++ counterpart object. Listing shows the implementation details.

Listing: method binding a C++ instance to a Python Reference Object.

```
C++ Plug-In Method:
SimulationService * cast(IService *);

C++ Interpreter Method:
IService * getService(const std::string &strNameInstance);

Python Binding:
#SimulationServicePtr: cast operator provided by SWIG which creates
#a Python reference given a C++ ptr instance
simulationService = SimulationServicePtr
(
    SimulationService.cast
(
    PythonService.getService( "simulationService")
)
)
```

This approach is necessary, as in contrary to C++, Python can not interfere directly with pointers but rather through references. Therefore, exposing C++ objects to the scripting interface require generating a special Python object that encapsulate the address of the corresponding C++ object. A Python class will act as a transducer, where each call will be propagated to C++ implementation. As we rely on SWIG, we can take advantages of its inner mechanisms that automate this process. The following listing describes a C++ class and its exposure in Python:

Listing: C++ class and its exposure in Python

```
C++ Class:
vhdSimulationService
{
    void addWayPoint(const std::string & strCharacterName, vhdVector3 vecPt);
    void clearSubGoal(const std::string & strCharacterName);
    void setCurrentState(const std::string & strName, const std::string &state);
    vhdVector3 getPosition(const std::string & strCharacterName);
    void setPosition(const std::string & strName, const vhdVector3 &vecPos);
    void setOrientation(const std::string &strName, const vhdQuaternion& quat);
    //...
};

Python Exposition:
import _pyvhdSimulationService
#...
class vhdSimulationServiceBody(_object):
    #...
    def addWayPoint(*args): return
    _pyvhdSimulationService.vhdSimulationServiceBody_addWayPoint(*args)
    def clearSubGoal(*args): return
    _pyvhdSimulationService.vhdSimulationServiceBody_clearSubGoal(*args)
    def setCurrentState(*args): return
    _pyvhdSimulationService.vhdSimulationServiceBody_setCurrentState(*args)
    def getPosition(*args): return
    _pyvhdSimulationService.vhdSimulationServiceBody_getPosition(*args)
    def setPosition(*args): return
    _pyvhdSimulationService.vhdSimulationServiceBody_setPosition(*args)
    def setOrientation(*args): return
    _pyvhdSimulationService.vhdSimulationServiceBody_setOrientation(*args)
```

Moreover, C++ is a complex language with many intrinsic features that are not directly available in Python. To overcome this limitation, a solution consist to expose at the scripting level, only high-level and flat APIs that do not feature pointers or complex data types. Sometime, alternative exist, such as for situations where pointers are passing through set of methods. The idea herein relies on storing the address of this pointer into a variable as illustrating in the next listing.

Listing: Python and passing through pointers

```

C++:
pointer *getPointer();
setPointer(pointer *);

Python:
pythonPseudoPointer = getPointer()
#the converter store the address
#can not use directly the pointer, due to lack of pointers support
#but we can pass through the address
setPointer(pythonPseudoPointer)
#the converter takes the address and cast it to destination pointer type

```

Finally, SWIG provides set of converters and wrappers around most C++ types, including STL instances. This way, a STL containers can be converted into a Python List or Map and vice and versa.

7.4.6 Python Module Registration

For exposing C++ modules to the scripting interfaces, we need to need to register them. SWIG provides a utility function that simplifies such registration. The registration will occurs if the end-user specify into XML-based configuration files if an active C++ module need to be exposed in Python using late-binding mechanisms. The following listing describes the module registration.

Listing C++ module registration:

```

vhdServiceHandleRef serviceHandle;
vhdServiceManagerRef serviceMgr = serviceContext->getServiceManager();
if(serviceManager->hasService("vhdPythonService", ""))
{
    serviceHdle = serviceMgr->getServiceHandle("", "vhdPythonService");
    if ( !serviceHdle || serviceHdle->isInitialized() == false)
        return false;
    PyEval_AcquireLock();
    init_pyvhdSimulationService();
    PyEval_ReleaseLock();
}

```

7.4.7 Performance

When working with an interpreted scripting language, some precautions must be taken. You have to understand the capabilities of each language. For instance, if you write Python code to do some heavy floating-point work and then compare the results to C++ you will be disappointed. Python is a slower language. Every variable reference is a hash table lookup, and so is every function call. This will never give performance that can compete against C++, where the locations of variables and functions are decided at compile time. However, this does not mean that Python is not suitable for 3DRTS, if used appropriately. If you are doing string manipulations or working with STL sets and maps in C++, then Python code may actually be faster [Bagley]. The Python string manipulation functions are written in C, and the reference counted object model for Python avoids some of the string copying that can occur with the C++ string class. Sets and maps are $O(\log_n)$ for most operations, whereas Python's hash tables are $O(1)$.

7.4.8 Memory Management

In Python, all objects are reference counted, which removes much of the need to explicitly manage memory. However, this does not resolve all memory management issues. For instance, if the simulation creates many short-life instances, the memory will become fragmented. To overcome this situation, we can isolate all Python memory allocations into a dedicated memory pool. Another memory issue may arise if some Python objects allocated globally keep a circular chain of references to other objects, preventing them from being deleted. Moreover, garbage collectors, such as the one built into Python, have unpredictable behaviors and can

produce inconsistent frame rate. It is generally better to control the garbage collection manually when processing time is available.

7.5 Concurrent Multitasking

Different heuristics for developing multitasking operations exist. In the language Python, several choices are available. The first category or real threads allocate a full-size stack for each thread of control, which is a source for overhead. The second category is refers as “users” threads and involve transfers of control. Generally, they are difficult to implement in a language implementation based on a single stack. The next sections will describe two structures from this category: the coroutines and microthreads [Hoffert1998, Dawson2001].

7.5.1 Coroutines and Microthreads

The development of correct multithreading applications is widely acknowledged as a complex task. It is desirable to greatly ease the development of cooperative multitasking, providing multithreading functionality to non-experts, while removing the difficult job of debugging race conditions and other data synchronization conflicts. The concept of a coroutine is one of the oldest proposals for a general control abstraction. It is attributed to [Conway1963]. The idea is to provide control behaviors in a concurrent context. However, general-purpose languages designers have disregarded the convenience of providing programmers with this powerful control abstraction. Recently, scripting languages like Python, Lua and Perl have investigated cooperative task management as an alternative to multithreading environments [Schemenaur2001, Adya2002, Bouvier2005]. A restricted form of coroutine in the form of simple iterators or generators is called a microthread. A microthread represents a unit of execution that has its own private data and control stack, while sharing global variables with other microthreads. Due to a cooperative concurrency model, microthreads must explicitly request to be suspended before another microthread may run. Within the Python environment, a microthread is a particular function, which contains a `yield` statement [Schemenaur2001]. When called, a microthread function must return a first-class object that can be resumed at any point in the program within the microthreads control stack. As microthreads do not suffer from expensive context switches, several thousand of them can run at a time without significant performance penalty [Tismer2000]. Microthreads allow greatly simplified AI and object update code [Carter2001] by moving much of the object state information into local variables, where it naturally belongs. Microthreads can be used to write functions that generate one result and yield back to the main program. The main program can then resume them later on, and they resume where they left off, with local variables intact. Since microthreads are based on cooperative multitasking, data synchronization is limited.

7.5.2 Microthread Manager

To simplify the use of microthreads for cooperative multitasking, we introduce a microthreads manager, which controls microthreads execution. This manager illustrated in the listing in section 7.5.2.1 is able to independently control each micro thread’s status (running, paused, or terminated). By communicating with the manager, an application can interact with the microthreads, changing their status on demand. During development, developers can break any scripts at run-time, which is useful for prototyping behaviors. In the scenario illustrated in Figure 7.1, we can observe that microthread id1 receives control and continues its execution where it last yielded. When a *yield* statement is reached, it gives control back to the manager. Simultaneously an external event requires the termination of the microthread id3. The manager will send this information to the microthread, releasing all its variables’ states. When this operation is completed, the microthread id3 will send a message to the microthreads manager, allowing the update of the current list of active and inactive microthreads. In the example, microthread id2 is currently inactive.

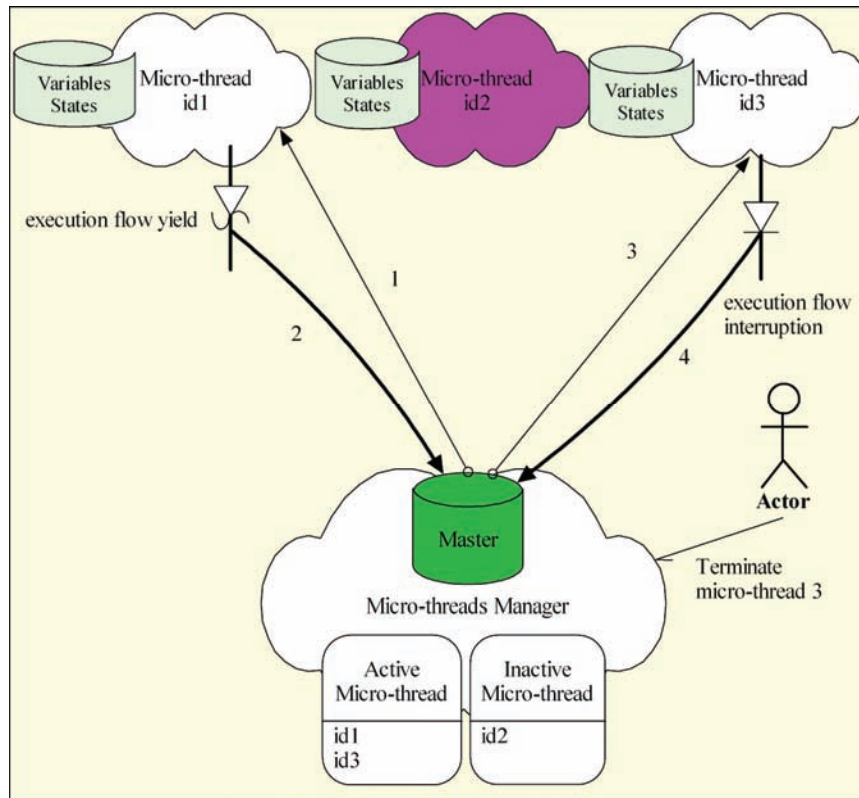


Figure 7.1: Microthreads execution model.

7.5.2.1 Implementation

The microthreads manager has to keep a list of microthreads with their status. It needs to handle creation, execution, and destruction of every active microthread. The listing above excerpt the key element of the microthreads manager implementation

Listing: Microthreads Manager

```
#The manager is itself a Python generator.
def pyMicroThreadManager():
    """
    to synchronize the Python micro thread manager with C++ code
    we yield at each iteration, leaving the C++ code triggering the
    .next() statement
    """
    while 1:
        yield 0
        if (isMicroThreadManagerPaused()):
            #so we pause this manager and wait until we resume it
            yield 0
        elif (isMicroThreadManagerMustDie()):
            #Break and terminate the microthread generator
            break
        else:
            #Safety removal of killed microthreads
            for i in range(len(g_pyMicroThreadToBeRemoveFromList)):
                removeMicrothread(g_pyMicroThreadToBeRemoveFromList[i])
            g_pyMicroThreadToBeRemoveFromList = []
            #We manage each microthread status independently.
            for i in range(len(g_pyMicroThreadList)):
                g_pyCurrentMicroThreadInRun=i
                nameOfMicroThread=g_pyMicroThreadNameList[i]
```

```

if (not isMicroThreadPaused(nameOfMicroThread)):
    #handle script calling C++ code throwing exceptions
    try:
        #if it is already finish will throws exceptions
        yieldResult = g_pyMicroThreadList[i].next()
        if isMicroThreadMustDie(nameOfMicroThread)
            or yieldResult==1:
            pyEndOfMicroThread(nameOfMicroThread)
    except ValueError:
        print "Safely terminate the microthread"
        pyEndOfMicroThread(nameOfMicroThread)
    #check is an external event require to terminated the manager
    yield pyEndOfMicroThreadManager()

#first create the manager
g_pyMicroThreadManager=pyMicroThreadManager()
#create the micro thread list
g_pyMicroThreadList=[]
#a parallel one with names not generator this time
g_pyMicroThreadNameList=[]
'''
we create a list of micro thread that have been killed, to be
safely remove in sync by micro thread manager
'''
g_pyMicroThreadToBeRemoveFromList=[]
'''
we create a variable which contains the ID in the list from the
last microthread called, in order to be able to remove it in case
of C++ throw Exception, we simply kill this microthread
'''
g_pyCurrentMicroThreadInRun=-1

```

7.5.3 Embedding Python

If the core engine API is exposed to the scripting language, complex scripts can be developed. A typical use case is to write algorithms that need to run fast in C++ and to control them within a script. As every 3DRTS has its own requirements, different configuration modes have been developed. All of them allow the script to execute in different ways.

7.5.3.1 Blocking Mode

In this mode, illustrated in Figure 7.2, the main thread waits until the interpreter finishes its update, which is performed using microthreads that are registered with the microthreads manager discussed above. The microthreads notify the microthreads manager of their status. The python microthreads update loop runs at a different frequency than the core C++ engine, allowing executing complex scripts without reducing the C++ core path performance.

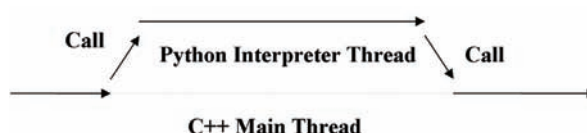


Figure 7.2: Case 1, blocking mode.

7.5.3.2 Cooperative Multitasking Mode

In this mode, illustrated in Figure 7.3, microthreads are used to run scripts via cooperative multitasking. This ensures that microthreads updates do not run at a higher frequency than the C++ code path, avoiding microthreads updates that run faster than necessary and use too many valuable system resources. In general, this is the most appropriate mode to use.

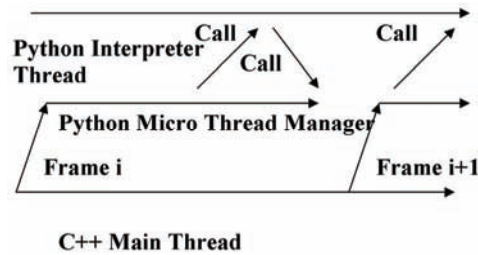


Figure 7.3: Case 2, cooperative multitasking.

7.5.3.3 Concurrent Multitasking Mode

In this mode, illustrated in Figure 7.4, the Python microthreads manager runs in a thread created within the Python interpreter thread. This clear separation between the C++ and Python code paths allow to prevent frame rate inconsistency when the overall scripting workflow differs on a frame basis.

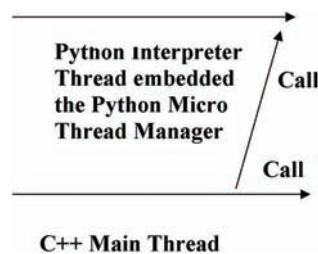


Figure 7.4: Case 3, concurrent multitasking.

7.5.3.4 Custom Preprocessor and Specific Keywords

One purpose of the system described here is to provide a higher-level of abstraction for microthreads usage. The concrete implementation must be hidden. The system should handle all variable and object creation and management. To facilitate the script development process, we developed a custom preprocessor. Its purpose is to translate simple scripts with custom keywords into native Python microthreads code. The use of custom keywords followed by an automated preprocessing step is meant to reduce mistakes made due to confusion when coding this logic by hand. This allows as well the modification of the manager without breaking hundred of scripts. The preprocessor also generates Python object references that bind to C++ objects at run-time. This removes the need to explicitly bind these objects into each microthread context. To simplify script creation, two new keywords have been created: the **vhdYIELD** and **vhdRETURN** statements. The **vhdYIELD** keyword gives control back to the microthreads manager that will do the transition to the next microthread in the stack. To qualify as microthread, scripts need at least one mandatory **vhdYIELD** statement. Using this keyword enables long-running routines to be distributed across several frames. Developers can set this statement in key areas within long or unpredictable algorithms like path planning request. The **vhdRETURN** keyword breaks and terminates the current microthread. By default, the last line of all microthreads must be the **vhdRETURN** statement. For illustration, let us investigate the script depicted above:

Listing: A Basic Script

```
B = 0
while (true):
    B=B+1
    if B == 10:
        vhdRETURN
    vhdYIELD
```

This sample script will be executed in ten consecutive frames. Every time the Python interpreter reaches the keyword **vhdYIELD**, the microthread will yield back to the microthreads manager, which will

eventually process the next microthread in the queue. When B equals 10, the execution of **vhdRETURN** will notify the manager that this microthread must be terminated. As these extra keywords are not parts of the Python language, it will not run within a standard Python interpreter. The custom preprocessor tool described above translates listing above into the native Python code as shown below:

Listing: Python compliant script based on preprocessing the script above.

```
#microthread definition, set an unique id for the generator's name
def __PythonScript2123():
    """
    We use a try and catch block to avoid that the script calls a
    C++ method to propagate the exception.
    We also dump the trace back helping the end-user
    """
    try:
        #the original script begin here
        B = 0
        while (True):
            B = B + 1
            if B == 10:
                #The preprocessor replace vhdRETURN with this Python code
                yield pyEndOfMicroThread(
                    "__generatorObject__PythonScript2123")
                #The preprocessor replace vhdYIELD with this Python code
                yield 0
    except:
        traceback.print_exception(sys.exc_info()[0],sys.exc_info()[1],
                                   sys.exc_info())
        yield pyEndOfMicroThread("__generatorObj__ PythonScript2123")

#Generate the Python generator
__generatorObj__PythonScript2123=__PythonScript2123()

#Register the microthread
g_pyMicroThreadList.append(__generatorObj__PythonScript2123)
g_pyMicroThreadNameList.append("__generatorObj__PythonScript2123")
```

The preprocessing system has its limitations. For instance, it will not be able to discover when a long-running main loop needs to be split over several frames. Alternatively, whenever the script author wants to explicitly terminate a microthread. Finally, from a run-time performance perspective, the impact of parsing the scripts occurs only during development. For application deployment, all the scripts are distributed as Python byte code.

7.5.3.5 Threading Management

The configuration modes provide the ability to run scripts in different situations such as cooperative or concurrent multitasking environments. The listing below illustrates the separation between those two modes from an implementation standpoint.

Listing: Script execution using either cooperative or concurrent multitasking mode.

```
/** register the vhdPython script as a python microthread
 * @param scriptName name of the script (UID)
 * @param yieldScriptText script text
 * @return the code execution
 */
int executeYieldScript(const std::string &scriptId,const std::string & script);

/* create a standard python thread */
int executeThreadScript(const std::string &methName,const std::string &script);

/** execute and create a standard thread or if the keywords vhdYIELD
 * or vhdRETURN is detected, create a python microthread
 */
int detectAndExecuteScript(const std::string&methId,const std::string &script);
```

7.6 Authoring Tool

To facilitate the development process, authoring tools are mandatory. In our system, a Python Console GUI (see Figure 7.5) based on a text editor provides the communications layer and interactions with the system. The editor gives feedback about the current state of each Microthread and uses color syntax highlighting not only for the Python keywords, but also for specific ISOs as well. Using this tool, end-users can load, write, save and execute vhdPython scripts. The vhdPython syntax is able to run all standard python scripts with the addition of specific features easing the management and creation of microthread. Within the editor, end-users can observe and change the status of each script (run, pause, stop).

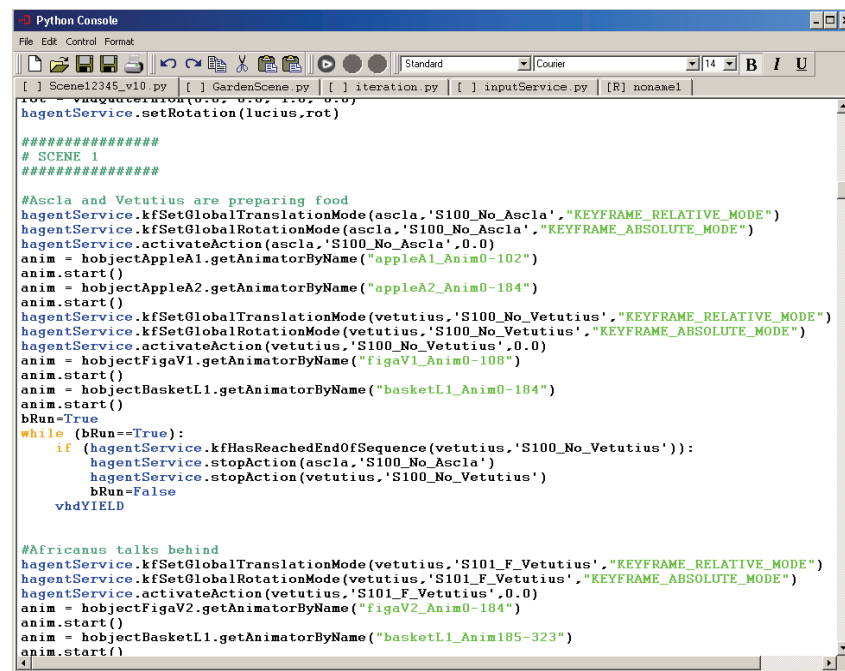


Figure 7.5: Python Console.

7.6.1 Features

The features integrated with the vhdPython Console include the following option:

- New document: open an empty document.
- Load / Save / Save as: load and save Python script either as into text file format or into Python byte-code.
- Print: print the current script.
- Script status: feedback information of the different state of each script
 - [] not running (inactive)
 - [P] script interrupted (in pause), can be resume at any time, and the script will continue where they left off, with local variables intact.
 - [R] script running (under the control of the python microthreads manager)
- Play / Pause / Stop buttons: interact with the script control flow.
- Code indentation: automate the code indentation that is critical in Python, easing the development of scripts.
- Color highlighting.
- Blue keywords define both specific features such as additional keywords and reference of C++ services expose through python. This inform the end-user if C++ modules where correctly exposed to the scripting interface.

7.6.2 vhdPythonService XML Configuration

A property factory accepting an XML as an input stream serves to configure the vhdPython Service as describe in the following listing.

Listing vhdPythonService XML configuration:

```
<vhdPythonServiceProperty name = "pythonService">
  <bPrintRedirection> TRUE </bPrintRedirection>
  <bSynchronizedPythonWithCPlusPlus> TRUE </bSynchronizedPythonWithCPlusPlus>
  <bSynchronizedPythonWithCPlusPlusButRunningInParallel> TRUE
</bSynchronizedPythonWithCPlusPlusButRunningInParallel>
  <bGarbageCollectorEnabled> FALSE </bGarbageCollectorEnabled>
  <iPythonThreadPriority> MIN_PRIORITY </iPythonThreadPriority>
  <strInitializationScript>initializationScript.py</strInitializationScript>
</vhdPythonServiceProperty>
```

The different parameters serve to modify the behavior of the internal Python Interpreter:

- **bPrintRedirection**: notify if the output from the Python interpreter will be redirected to the standard C++ output. By putting this flag to false, the interpreter run slightly faster with the lost of the interconnection of the Python and C++ outputs.
- **bSynchronizedPythonWithCPlusPlus**: if true means that the Python Microthreads Manager runs at the same frequency than the main C++ thread. This is useful when we do not want that the scripts run faster than the core engine.
- **bSynchronizedPythonWithCPlusPlusButRunningInParallel**: this option is ignored if the previous tag is set to false. When actif, this option notify that even so we are keeping the same frequency between the Python interpreter and the main C++ thread, the execution of both C++ and Python code will be paralyzed. Even so, it require one additional thread, this avoid that CPU do not run at full speed due to I/O operations affecting scripts. In addition, it gives the opportunity to skip a frame for the scripting interface, reducing unpredictable situations, when a script require too many processing resources or freeze the C++ code path. By disabling this option, the system reduces the number of threads by one, freeing some CPU cycles for context thread switching.
- **bGarbageCollectorEnabled**: specify if the end-users want to control the garbage collector by setting manually the collection of unused memory blocks or if the system will use the default Python garbage collector.
- **iPythonThreadPriority**: specify the thread priority for the Python interpreter. This gives the opportunity to adapt the resource allocated for running scripts. For intensive 3DRTS, the system can lowered the time dedicated to handle scripts and events.
- **strInitializationScript**: specify the first script that will be executed. This may serves to specify global parameters or to import specific Python modules. In effect, variables declares within a Microthread will remain local.

7.6.3 Late Bindings

Our architecture provides mechanisms to automate the bindings of vhdService reference objects into python reference objects. This facilitates the use of scripts by assigning python reference variable automatically. Using XML tags such as the one describe on the listing below, users can ask the system to generate those objects on the fly.

Listing XML for Python Reference Object Bindings

```
<vhdPythonModuleLoaderProperty name = "viewer">
  <strModuleLoader> vhdOSGViewerService </strModuleLoader>
  <strPythonScript> pyvhdOSGViewerService.py </strPythonScript>
</vhdPythonModuleLoaderProperty>
```

7.7 Special Use Case

Embedding an interpreted scripting language goes beyond driving the data through scripts. Scripting languages such as Python features intrinsic functionalities that facilitate the management of assets. For instance,

Python can natively serialize data structure into files. Python also comes with mechanisms to restrain the access rights to specific files and functionalities. This avoid that hackers attempt to modify system files through the simulation by uploading scripts and prevent users to access low-level routines exposed only for developments. The next sections will introduce some particular use cases using our extended version of Python.

7.7.1 Safe Sand Box

The dynamic nature of data-driven architectures systems open a vast opportunity for non-programmers to interact with the system [Poiker2002]. The system maintenance and evolution increase the inherent dangerous practices for resolving specific issues. The code base on the script level implies that the code is not a stable and will change often. The scripts represent the communication layers for sending simulation events. In addition, they control the simulation from higher perspectives allowing modifying intrinsic parameters that need to be constantly changed for fine-tuning simulations. The advantages of offering such capabilities are notorious and allow a wide scope of users, which may have limited technical experience to implement scenarios and try out ideas. Generally, they many not have the skills and experience to produce robust and reusable code. This leads to apply techniques that restraint the capabilities on the critical systems infrastructure. Typical situations cover the case where the user either could introduce critical bugs or could operate on sensitive subsystems like computing files manipulations. A solution for this problematic is the safe sandbox paradigm which was first introduced by [Walker2003]. The conceptual idea behind this notion is to keep all the functionalities provided by the scripting language but to restraint its usage and access to clear defined packages. By default, a standard scripting interpreter could operate on all the available functionalities including I/O operations. In the situation where the application is supposed to run on dedicated server, any end-user could potentially remove sensitive files from the hard drive. Thus being able to limit the access on specific folders or packages avoid that unaware simulations designers introduce undesirable instability within the system. Fortunately, Python is offering a set of facilitators with the appropriate mechanisms.

7.8 Optimization

Interpreted scripting languages are slower to execute the code than compiled language such as C++. Developers should not use scripts for time-critical or heavy floating processing code sections. However, this should not prevent the integration of mechanisms that would improve the run-time performance of our scripts. The following elements represent functionalities that do not require more efforts for developers but that can increase the processing throughput of scripting languages.

7.8.1 Garbage Collector

Garbage collector is a powerful mechanism that frees developers to handle memory management. In Python, the garbage collector generally call the collector every few seconds (~5 sec) with its default behavior. This may lead to inconsistent frame rate as the interpreter need to allocate some resource for freeing memory blocks. In the worse case, spike in the frame rate can be observed every time the garbage collector is active. To minimize this side-effect, a solution consist to disable the automate garbage collector and to call it when processing time is available. For instance, the garbage collector can be disabled during at run-time and the programmers can manually collect the data during the transition of environments or whenever the application is paused. As an outcome, the application has a smoother and more consistent real-time behavior

7.8.2 Stackless Python

Stackless Python [Tismer2000] is an enhanced version of Python increase the built-in feature of the language but also escape the limitation of 4k required by the default Python environment. In addition, the memory footprint is smaller and the run-time performance is 15-20% faster than the standard Python interpreter is. More interesting, Stackless Python provided a more advanced implementation of Python Microthreads. Within this environment, it become possible to wrap functions as microthreads, to take advantages of a built-in schedulers and also to serialize tasklet to disk through pickling for later resumption of execution.

7.8.3 JIT Compiler

The Python interpreter can be equipped with a JIT compiler or Just in time compiler is (beside the interpreter) as part of the virtual machine and accelerates the program execution by a factor of 2-4X. Just in time means "straight in time". JIT compiling is a technology from practical computer science, in order to improve the performance of application software, which is present as byte code. The JIT compiler translates at run time if necessary the byte code into a native machine code (thus a code, which the processor can process directly). Highly developed JIT compilers can generate particularly for dynamic languages faster code than conventional compilers, since they can to meet and dynamic optimizations.

7.8.4 Byte-Code

For reducing the time to parse and compile Python scripts at run-time, it is possible to store these scripts into binary byte-code format, that is, a sort of intermediate code that is more abstract than machine code. The Python byte-code being interpreted by the virtual machine, it remains portable. As an outcome, the same files can be executed across different platforms or architectures. This is the same advantage as that of interpreted languages. However, because byte-code is usually less abstract, more compact, and more computer-centric than program code that is intended for human modification, the performance is usually better than mere interpretation.

7.9 Experiments

So far, the microthreads management module has been used for controlling dynamic components such as cloud simulations, particle systems and within agent simulations (sees Appendix G for concrete examples). However, the system is far from being perfect and comes with some limitations, notably coming from the Python microthreads model. For instance, it is not possible to interrupt a microthread outside its running context. Consequently, specific handling may be required in some circumstances. As a side note, the actual implementation of coroutines in Lua provides such functionalities [de Figueiredo2006].

For analyzing the scalability of this architecture, several different test bed applications were built to test its viability for use in crowd simulation. One test-bed features 800 individual agents running on dedicated microthreads. The purpose of this application was to control the agent's behavioral context objects, where each entity keeps its own internal logic within a microthread. The distinct separation in the entity representation provides a granularity on the entity level. Thus, the AI engine is able to control the scheduling of every active entity upon the real-time constraints and change their inherent logic based on their current LOD. It also provides higher flexibility in the flow execution.

Different use cases have served for benchmarking the system. Tests have been executed on a Pentium IV Xeon @2.2Ghz with 1 GB of memory and an Nvidia Quadro Pro graphics card. The Figure 7.6 shows the performance impact of the major components used in this simulation. The 3D rendering component is the more expensive task. The animation and AI update are related to the number of active entities, with the difference that non-visible entities animation update remain a constant time based functions (e.g., only the skeletal root orientation and position are modified). The Appendix G presents different concrete examples of the use Microthreads for controlling the simulation events.

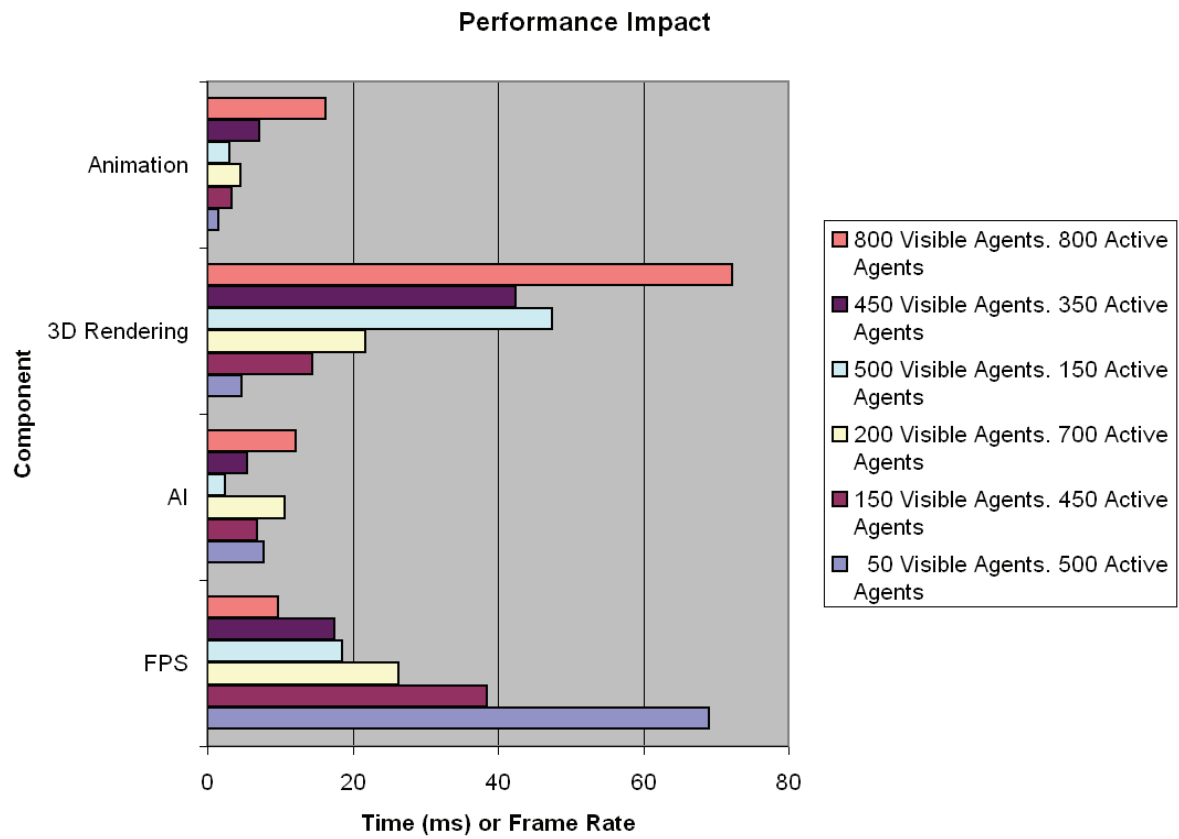


Figure 7.6: Performance impact.

7.10 Conclusion

In this chapter, we have demonstrated a new approach for handling multitasking operations in a safe environment. This gives to designers the ability to create separate flow of control for each character with minimal performance overheads. This usage of microthreads as lightweight threads makes this approach also suitable for single processor hardware, offering high-level interactions with the system functionalities at run-time.

Chapter 8

Data-Driven AI Engine

Until recently, crowd simulations were mainly confined for non real-time applications. The management of thousand of dynamic entities and their associated requirements were too important for being handled properly by the existing systems. The improved processing throughput currently available allows producing realistic crowd simulations within real-time constraint. A purpose of developing crowd simulations is to create more compliant and populated virtual worlds. These offer a better representation of the simulated worlds and serve as trying to understand how human being reacts within a crowd. The first iterations of AI engine integrate within 3DRTS were under delivering as the technical constraints did not allow researchers to exploit their ideas. With time, AI systems became more practical, their goals were to simulate and react to scripted scenarios. The AI branches were separated between techniques and methodologies with mainstream techniques like FSM [Khoo2002, Anderson2003, Fu2004, Shaw2004] to more complex approaches like neural network or genetic algorithms. Researchers working in these aspects do not consider that commonplace methods remain in the AI fields [Brooks1999]. However, these separations and the particularities of AI systems have prevented so far the development of industry standards like the one available for 3D graphics rendering, even so recent initiatives such as [IGDA-AI] tend toward this goal.

8.1 History of 3D Real-Time AI

Historically, the ideas of implementing AI systems were the results of the intensive work made by intelligence services for cracking enemy codes during the World War II. At the same period, Walter Pitts and Warren McCulloch were constructing neural network computations. In the 1950, Alan Turing was proposing a series of tests to identify if a computer was intelligent. For helping AI developments, different programming languages were developed like LISP [Stottler2003]. The next evolution of AI systems was to extend the definition by defining that AI systems should not only being able to answer question but should also perceive its environment. The large scope of AI management has prevented so far standardization for 3DRTS AI. If 3D renderers may follow similar methods for handling the rendering of fast 3D graphics using common schemes such as LOD-generation, culling and drawing steps, this is far from being the situation for AI components. The elaboration of an AI toolkit would allow researchers and game developers to develop ideas on the same environments. As the AI field requires testing intensively algorithms, such toolkits should be mostly driven by its data. In the early age of AI systems for 3DRTS, the AI resources were limited and where not suppose to be changed on demand. Some of the first attempts to provide the ability to modify the behaviors of virtual entity come with the apparition of games SDK allowing to modify game contents and events [MacLellan2000].

As of today, the gap that separate the AI research and game development technologies [Laird2000] is slowly reduced as current platforms feature enough processing power to handle research AI algorithms inside 3DRTS. Going beyond these constraints, 3DRTS are opting for integrating more realistic behaviors for improving the believability. In the past, interactive simulations were trying to simulate simple behaviors that can be seen as faked solutions. The 3D realism of current scenes required to match these visuals with more clever agents. Inaccurate behaviors are more visible with the increase quality of sound and graphic environments. Some initiatives, such as [Atkin2001], [Baekkelung2006] or [IGDA-AI], try to combine the two fields into a combined effort. Coming out of this thinking, the goals is to standardize common set of functionalities or SDK that will allow AI research community and game developers alike to shared their developments. Such common standardization need low-level optimizations but also to let AI researchers prototyping and analyzing algorithms both for narrow situations and within real simulations constraints. The benefits are obvious as it provide the same core of functionalities that graphic or sound programmers have access. The ability to built custom extensions based on a standard SDK would improve the AI developments by offering efficient environments and authoring tools. One advantage is to spread AI development to non-technical people through consistent tool chain. In the long term such approaches is required for commercial products and are slowly appearing through middleware [Dysband2003].

AI researchers may also directly benefits from standardizations, so they can focus on developing new ideas rather than focusing their efforts building an AI library re-implementing all the same functionalities such as

FSM. In addition, working on the same SDK, greatly improve the ability to analyze the performance of their system. In many occurrences, it is very difficult to analyze few routines within a running application. Thus, by developing on top of the same standardized SDK, researchers will be able to compare different approaches validating their results. One might ask why such standardization still does not exist. The AI field does not have strict objectives and criteria. If the purpose of 3D APIs is to render complex geometry as fast as possible, the results in AI depend on the requirements. It is not possible to decide which criteria will define that one algorithm is appropriate for all situations. Moreover, AI benchmarks for measuring the performance are more subjective than statistical. Therefore, the approach promote by [IGDA-AI] is to provide set of common tools and interfaces. The AI field relies on knowledge and such API should be versatile to accept the different interactions that simulations gender needs. The good balance between what should be part of the standard from what should remain separate is complex. If too many elements are part of the standard, the potential the APIs became overloaded is important but at the same time, a too focus standardization will limit the usability. The specifications of a usable and powerful API for AI system is difficult as the number of abstractions and communications layers that affect both low-level and high-level components is important. Here, the importance of authoring tool is critical. To visualize AI behaviors, developers need to conceive tools such as the MissionLab [MacKenzie1997], the BrainFrame [Fu2002] or the vhdCrowd Rule [Ulicny2005] editors. Non-AI middleware are also promoting authoring tools for creating agent's behaviors [Softimage-AI, Virtools_AI].

Most AI implementations in 3DRTS do not pretend to provide algorithms that simulate intelligence and thinking to a level close to humans. The 3DRTS constraints have forced researchers to develops methodology that provide AI features which are convincing enough for the simulation purpose. The research have resulted in commonplace methodology for describing AI systems based on individual agents like finite state machines, goal-based system or planning [Stout, Fairclough2001, Buckland2002, Tozour-AI2002, Champandard2003, Sweetser2003, Funge2004, Rabin-AI2004, Schwab2004, Yiskis2004, Buckland2005, Jones2005]. Most current interactive simulations features some level of AI managements and few of them are developing more complex AI [Evans2002, Ward2002, Valdes2004, Bishop2005]. In addition, AI management is not restrained to dynamic entities but also affect camera navigation [Haigh-Hutchinson2005].

AI techniques uses in 3DRTS rely on the combinations of well-defined AI methodologies that include FSM, fuzzy logic, goal-oriented or genetic algorithms. The current challenges face by AI researchers is to develop an engine that can be extendible efficiently. This is primordial as the requirements of AI computations are sensitive to the simulation contexts. The same behavioral rules may not be adaptable from project to project but their internal logic remains similar. For instance, in the AI engine presented in [Ulicny2005], all the FSM states are compiled as C++ source. This prevents to modify dynamically virtual humans behaviors. The next sections will highlight a fully Data-Driven AI engine architecture. From all the different AI techniques developed, the most common techniques ensure the system will remain in strict boundaries. It may seem not relevant to restrict the AI computations but many simulations are not designed to handle nontrivial scenarios. Therefore creating an evolving system should be carefully planned. In fact, we can separate techniques into different categories, based on their constraints in the AI management.

8.1.1 Restrictive AI Techniques

When developers design interactive simulations, they want to emphasis on different aspects. For the creation of compelling virtual environments, storytelling is an important element. There is a need to control the simulation events. To venture into techniques that may have unpredictable results may reduce the experience. Thus, the use of techniques working within defined boundaries is clear advantages. For instance, FSM allow dividing a game object behavior into logical states, which are interconnected to each other. This allow to assign one state for each simulated behavior and their transitions. The simple management and creation of FSM as well as their ability to resolve many problematic continue to make this methodology popular. One severe limit with the use of FSM is that they do not suit well with large datasets, making their maintenance difficult tasks. Some variants of FSM like HFSM and FuFSM allow extending their usability into more specialized components.

8.2 AI Engine

For being able to create characters behaviors, we need a combination of the techniques describe in section 8.1. The literature refers such package as AI engine. An AI engine provide the abilities to create and control the entities in real-time [van Lent1999, Kuruszewski2004]. Until recently, AI systems were confined to

relative simple routines that were built at the last minute. The processing power and efforts dedicated for their creation were restraints. Hopefully, different initiatives tend to provide more complete solutions under the form of library [Fear, Soar] or full middleware solutions. Middleware software dedicated for AI managements become available and are used in successful products [BioGraphics, Radiant, RenderWare_AI]. The particularities of AI developments about specific requirements are a limitation in this expansion.

8.2.1 Core Principle of AI Systems

AI systems can be seen as actually being two separate areas:

- **The Logic layer**, also known as the “*engine*” level. This is where the underlying systems that control AI characters within the game world are placed. This would include components like animation selection, path finding, obstacle avoidance, and the like.
- **The Data layer**. Represent the layer where the actual AI behaviors and decision structure to move and progress within the simulation are described. At this level, we are describing element like specific moves to perform, where to travel to, or which object to use.

Many AI systems today still rely on entirely code based implementations, or what we refer to as the “*monolithic architecture*”. In these systems, both the logic and the data layers are both written in code. Any change at the data level would require a system recompilation. This leads to a programming bottleneck in development, since a programmer is required to not only add to the system, but to rebuild the system so changes can be tested. In addition, as the system scales upwards in complexity, a monolithic system becomes more and more unmanageable and inefficient. To prevent this, we must enforce a clear separation between these two basic AI levels. Each level will then be easier to implement and improve independently, as well as allowing for greater synergy between the two. If we perform this separation by having the data layer be defined by configuration files and/or scripts, then this is what is widely known as Data Driven architectures [Grimshaw1989, Bilas2002]. They have been proven to provide more streamlined and effective results [Shumaker2004]. One of the basic problems any data driven system needs to address is manipulating flow of control from within the data. Our approach is built around a multilayer system, emphasizing minimization of functionalities at the base level and allowing for dedicated, plug-in extension modules to take care of specific cases.

8.2.2 Related Work

In the last couple of years, the introduction of programmable Graphical Programming Units (GPUs) and therefore the advent of programmable shaders for real time graphical applications [Lindholm2001] has shown that with relatively little efforts, great advances in the graphical quality of the average game can be achieved. The introduction of higher level programming languages for creating these shaders (for example: [Cg, GLSL, HLSL]) has demonstrated that even better graphical quality is reachable by providing more powerful tools directly to the developers [Fernando2003]. Through the same mechanism, further improvements in the quality and complexity of artificial intelligence could be achieved with a higher level of abstraction in the AI definition language. Many researchers are convinced that dedicated hardware for AI accelerators will become available [Funge1999]. Another possible inroad into this might be future hardware with multi-core processors, which might allow AI applications to have a dedicated core. These kinds of hardware evolution will allow for the creation and visualization of large crowd simulations with much better behavioral fidelity.

In the past years, off-line techniques have been elaborated in commercial systems [Koeppel2002, Moltenbrey2004]. Now, researchers and middleware companies are trying to integrate off-line techniques within interactive applications. [Scott2002] proposes architecture for managing AI in games. [Champandard2003] has developed the FEAR framework that allows for the creation of AI behaviors through XML configuration files. [Kruszewski2005, 2006] describe their middleware AI Implant and its integration in the next generation of gaming platforms. Similar data driven architectures like the one from [Bilas2002] have been exploiting the approach for commercial products as well. [Borovikov2005] explores a methodology that he refers as an Orwellian State Machine (OSM). OSM consists of a command hierarchy, messaging and FSM mimicking a bureaucratic dictatorship. Borovikov approach helps in the analysis and practical design of particular AI subsystems on both the individual and group layers. More recently, [Garces2006] propose an architecture for executing AI management in parallel using multiple threads. He describes several approaches for minimizing

the dependencies and locking issues for collaborative behaviors management. [Gilgenbach2006] present an alternative approach for sharing behavioral states by decoupling the logic and data layers. They do not explicitly record state transitions into a table but extend the behavior with “*runnable*” condition and priority, which determined the state transition at run-time. [Johnson2006] describe a goal-oriented AI architecture. Each goal can set up one of more sub goals to achieve its aim. His approach is a compromise between classical FSM and full planning system. Similarly to Johnson, our approach offers the abilities to generate sub-goals for path following or sequence of actions.

8.2.3 AI Engine Architecture

The system consists of multiple abstraction layers connecting every agent with both low-level (animation, physics, steering, etc.) and high-level functionalities (tactical senses, perceptions, memory, etc.) through consistent plug-in API interfaces [Mandel2004, Garces-FOCA2006]. By separating the core AI engine from the other plug-in modules; we increase the possibility to reuse the core engines for different projects. In fact, every plug-in module comes with their own set of constraints and specifications. They can be designed for speed, memory performance, high fidelity rendering, or whatever else you require. Because of this level of flexibility, it gives you more freedom when choosing the proper module based on the current project requirements (due to license fees, scalability, platform, etc.). It becomes possible to develop different kinds of real time applications using the same core AI functionality.

8.2.3.1 System Overview

Managing a crowd simulation with hundreds of highly customizable distributed entities requires constant modifications and adjustments in the behavior rules to produce convincing results [Sung2004]. Every agent must be able to accomplish different tasks like walking or interacting with environmental objects. In order to define the agent’s behaviors in a more intuitive way, you must use a higher level of abstraction, and defer the details to a lower level system [Fairclough2001]. In our system, this will be achieved by taking advantages of a scripting language for managing behavioral rules through prioritized tasks [McLean2002] and hierarchical Fuzzy State Machines (FuSMs), which will be explained in detail in the next sections.

8.2.3.2 Behavioral Engine

The system architecture relies on C++ templates using metaprogramming techniques as described in [Abrahams2004]. Each entity in the system is derived from a common core of template classes. They are combined using both inheritance and delegation through the property design pattern [Gamma1995]. Most entities share similar properties and derive from a movable property (for steering behaviors [Reynolds-Steer1999]) and a physical property (which contains such things as mass, gravity and the like). The engine works on the entity’s separate properties using common interfaces, to facilitate the plug-in functionality. The listing above shows a typical interface code snippet, in this case for the IAnimation class.

Listing: A snippet of the Animation Pluggable Interface

```
class IAnimation : public IPlugIn
{
    void defaultPosture() = 0;
    //key_frame
    kfActivate(const std::string& keyframeName)=0;
    kfSetTimeScale(vhtReal rTimeScale) =0;
    // ...
    // Walk Animation
    void walkSetDesiredFrontalSpeed(const vhtReal& rSpeed) =0;
    void walkSetPositionReachedTolerance(const vhtReal& rValue) =0;
    void walkSetDesiredPosition( vhdVector3 vecPos) =0;
    // ...
    //look
    void lookSetTargetLocation(const vhdVetor3 & vecPos) =0;
    void lookSetEyeBlinkParameters(vhtReal rCloseEyeAngle, vhtReal rPeriod,
                                   vhtReal rVariance) =0;
    // ... other actions ...
}
```

};

When a new behavior goal is initiated, the request starts at the top level of the system. For instance, the animation module might indicate that it desires to move an arm to a certain position. The “*animation module*” in this example is actually the high-level and common interface level. The actual work will then be done by the plug-in implementation underlying the animation system, which we call the “*concrete implementation*”. In this way, the AI engine is not bound to a dedicated animation or physics package. Each package is free to implement the common interface requests as it sees fit. A walking animation can be computed either using key frame animation or using physics based dynamic walking. Figure 8.1 shows an overview of the architecture, clearly displaying the separation of the plug-in implementations from the AI engine. The top layer contains the *iCharacter* entity, which encapsulate the different modules affecting an agent (behaviors, animation, and motor). In addition, a connection bridge ensures that each entity owns its encoded private behavior rules, and is in direct control of its concrete instance.

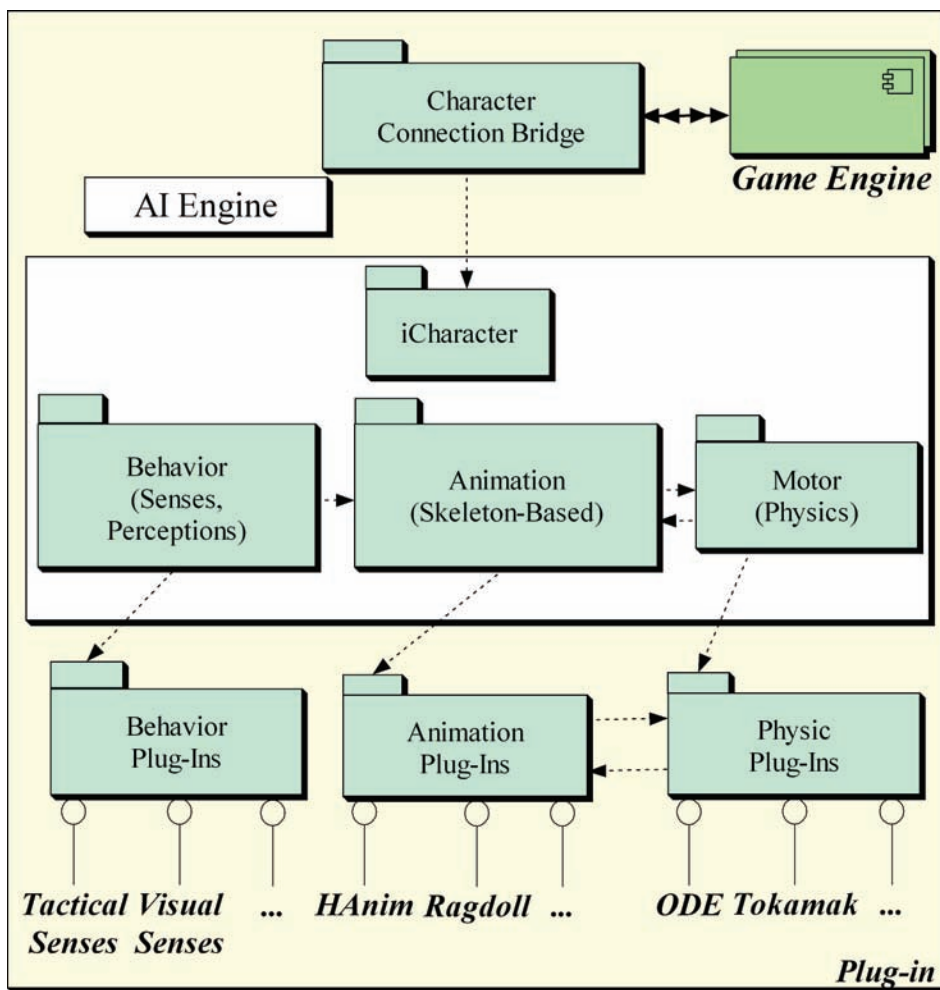


Figure 8.1: Hierarchy of properties and concrete body instances.

8.2.3.3 Data-Driven Classes and Properties

In our design, the information required to update the state of each agent must not depend on the data itself. These different abstraction layers ensure this separation (see Figure 8.2):

System Overview

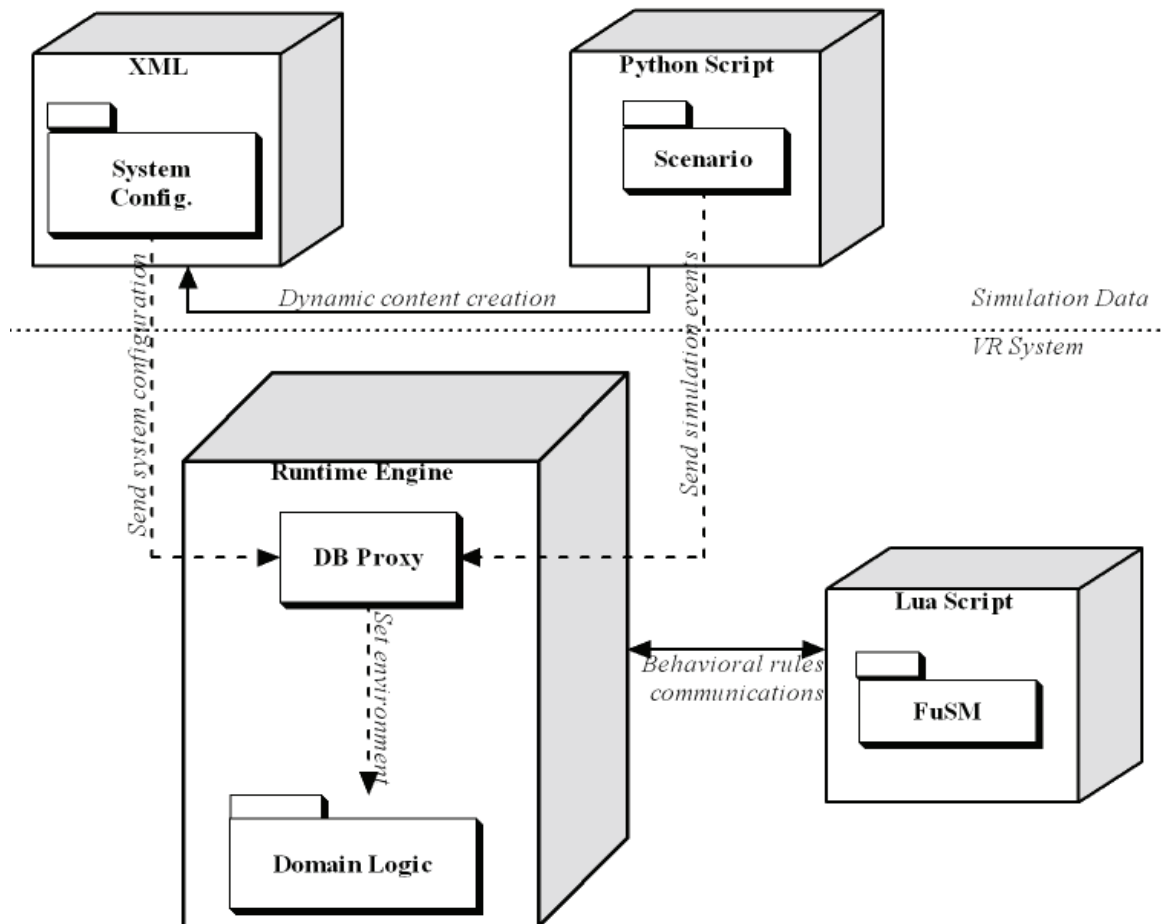


Figure 8.2: System overview of the multi-layer architecture.

- The behavioral rules and FuSM states are defined using Lua metatables [Ierusalimschy1996]
- Event propagation and world interaction is managed through Python microthreads [Python, Hoffert].
- Initial world description is set using XML configuration files.

At each abstraction layer, we take advantage of each language's unique peculiarities. The logic layer is written in C++, and the data manipulation is done using scripting languages [Yiskis-FSM2004]. The decision to use two different scripting languages is because of several reasons. First, Lua is generally a faster interpreted language than Python [Bagley2005]. Second, Lua and C++ can interoperate on a lower grain than using Python [Ierusalimschy2003]. Therefore, we use Lua for manipulating and updating the different state machines for each entity, so we can maintain a level of efficiency while keeping a data driven approach. Python, however, comes with much more third-party components, notably for database and network management, that we can leverage. Although the system is written such each scripting language does not interfere with the other, direct communications between Python and Lua is possible using the two-way bridge solution from [Niemeyer].

8.2.3.4 Hierarchical FuSM (Fuzzy State Machine)

FuSMs are a permutation of regular Finite state machines (FSMs), which use fuzzy logic rules instead of Boolean logic for decision making [Kantrowitz1997]. As a result, fuzzy states are not limited to being on or

off, but can hold an intermediate value. A fuzzy state system, unlike a finite state machine, can have any or all of its states on to some degree. While this curious nature makes constructing FuSMs slightly more complicated than creating a vanilla FSM, the existence of simultaneously active states reduces the predictability of the resulting behaviors. It also dramatically reduces the complexity of the state machine, as a wider range of different behaviors can be encoded with fewer states.

In our case, the FuSMs states and substates required by the application were defined within Lua metatables. In Lua, a metatable is an ordinary Lua table, but with extra specifications to its behavior (and how it uses user supplied data) under certain operations. In our case, these involved representing specific rules for dedicated states, allowing for scenario based rules (different environments, number of entities, etc.), and for providing reference access to the C++ character entity data structure. As shown in the listing above, each state description requires four keys to be provided:

- *Name*: Simply the name of the state.
- *Enter*: The function for the entity to call upon entry to this state.
- *Execute*: The updates function for the state.
- *Exit*: To be called when exiting the specific state.

Listing: A FuSM State describe in Lua metatables

```
-- create the State_PathBehaviorActivateGoal state
State_PathBehaviorActivateGoal = {}
State_PathBehaviorActivateGoal["Name"] = "PathBehaviorActivateGoal"
State_PathBehaviorActivateGoal["Enter"] = function(character)
    -- nop
end
State_PathBehaviorActivateGoal["Execute"] = function(character)
    character:walkTo(character:getDesiredPosition(), true)
    character:activateWalk()
    character:getPath():setNextWaypoint()
    if character:getPath():finished() then
        --done
        character:getFSM():changeState(State_PathBehaviorDone)
    else
        character:getFSM():
            changeState(State_PathBehaviorSelectGoal)
    end
end
State_PathBehaviorActivateGoal["Exit"] = function(character)
    -- nop
end
```

8.2.3.5 C++ and Lua Metatable Bindings

The core C++ engine contains a Lua interpreter and exposes its functionality using the [Luabind] library to generate “glue” code (a term that refers to the code required to manage variable and/or class bindings between C++ and Lua). Luabind shares many things in common with the metaprogramming approach of the Boost library, as it offers a set of functions that expand the Lua scripting language with extended object oriented mechanisms. This allows Lua classes to derive from other Lua or C++ classes. Since Luabind uses metaprogramming, it does not require a preprocessor compilation pass like the popular tool from [Manzur]. The compiler simply generates the glue code automatically. However, the compilation of the binding code can be relatively slow. One trick in fighting this is to keep the binding code in separate object files, instead of one huge binding depository. The next listing shows an example of C++ classes being exposed to Lua using the Luabind library.

Listing: Luabind Bindings

```

void iCharacter::registerWithLua(lua_State* pLua)
{
    //register the class into the module vhdEl
    luabind::module("vhdEl", pLua)
    [
        //indicate that the resulting Lua class
        //inherit from the exported base class
        .def(luabind::class_<iCharacter,
            luabind::bases<vhdEl::iBaseEntity> > ("iCharacter")
        .def(luabind::constructor<const std::string&>),
        .def("setWalkStyle" , &vhdEl::iCharacter::setWalkStyle),
        .def("setWalkSpeed" , &vhdEl::iCharacter::setWalkSpeed),
        .def("lookAt" , &vhdEl::iCharacter::lookAt),
        //...
        .def("performSensing" , &vhdEl::iCharacter::performSensing)
    ];
}

```

As you can see in previous listing, many aspects of the engine can be exposed to the scripting language. In this system, the behavioral rules and FuSMs that drive the animation and entities reactions are described within the Lua scripts. Every entity's state machine keeps an internal `Luabind::object` facilitator, which connect Lua objects and their C++ counterpart. This connection variable gives the user a straightforward hook to interact between the engine code and the Lua stack. By having each state containing its own binding facilitator, the system is not affected by the size or complexity of the FuSM for access time. The engine can execute and update the agent's state without knowledge of the state itself. The listing above shows a small snippet of the state execution controlling the logic within the engine.

Listing: C++ method for updating the active state machine

```

void FSM::update()
{
    //luabind object point to the current Lua metatable
    if (_luaCurrentState.is_valid())
    {
        //_pOwner: pointer to the entity
        _luaCurrentState.at("Execute") {_pOwner}
    }
}

void FSM::change(const luabind::object& newState)
{
    //call exit on the active state
    _luaCurrentState.at("Exit") {_pOwner};

    //change state
    _luaPreviousState = _luaCurrentState;
    _luaCurrentState = newState;

    //call the entry method for the new state
    _luaCurrentState.at("Enter") (_pOwner);
}

```

8.2.3.6 AI Architecture Using Prioritized Tasks

The performance impact of using a scripting language can be minimized in many ways. One-way is to use a manager that handle the workflow of the FuSM system by assigning tasks among categories of states, thus avoiding updating unnecessary tasks. However, as the engine is not bound directly to the data, the designer must tag the different states with relevant category information to facilitate this.

To keep real time performance high, any AI system needs to work within a budgeted period. It should constrain the number and/or complexity of the behaviors within the system to accommodate the available CPU

time [Funkhouser1993, Alexander2002]. This should keep the AI system requirements relatively stable, so as not to affect the immersion through inconsistent frame rate caused by spiking CPU needs [Schertenleib2002]. By providing a high-level approach that allows the designer to organize the different tasks within the system, we are able to separate each task's classification from the engine.

Among all the tasks that can be activated on a particular frame, we need to distribute their dispatching so we can ensure that the per-frame workflow remains constant. To do this, we separate the different tasks into three categories: those that need to be updated every frame, those that can be updated on a periodic basis, and finally those tasks that will be executed only if processing time is available.

You might check certain tasks on a periodic basis for different reasons. For checking if a key frame animation has completed you might use a periodic priority because it is unlikely to change on a frame-by-frame basis. Another reason to check certain tasks periodically is to introduce some of delay before the task propagates; such latency increases the sensation of AI presence by avoiding that the entity reacts immediately to every stimuli [Laird2001]. The amount of tasks that need to be updated each frame needs to be minimized, as this subtracts directly from the total time available in your AI performance budget.

By using categorized tasks, we can ensure the engine will maximize efficiency. We do this by using configurable search rules (custom to each category of task) to help direct our search through each bank of behaviors. We can ensure that more import tasks happen first, but that older tasks are not starved out of the system. Figure 8.3 shows a very simple overview of the Task Scheduler.

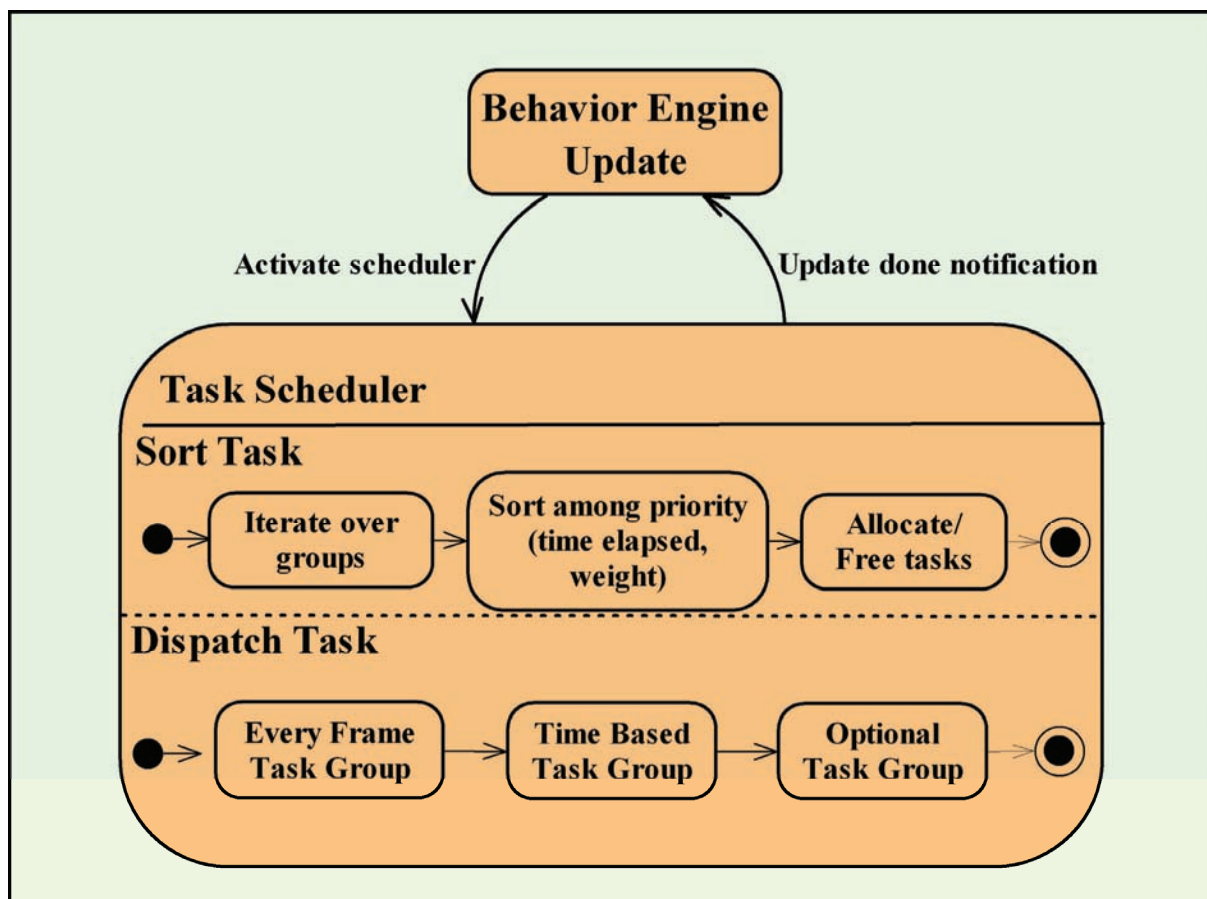


Figure 8.3: Tasks scheduler management.

8.3 Behavior

In our approach, the behaviors are defined into Lua Metatables defining the HFSM. Depending of simulation events, a behavior, or state is assigned to an agent. These roles are to orchestrate the simulation by affecting basic actions to agents. The different states include emotions (sad, happy, laugh). They also allow performing motion control (walk, look, hear...) as well as path planning request (see Figure 6.23).

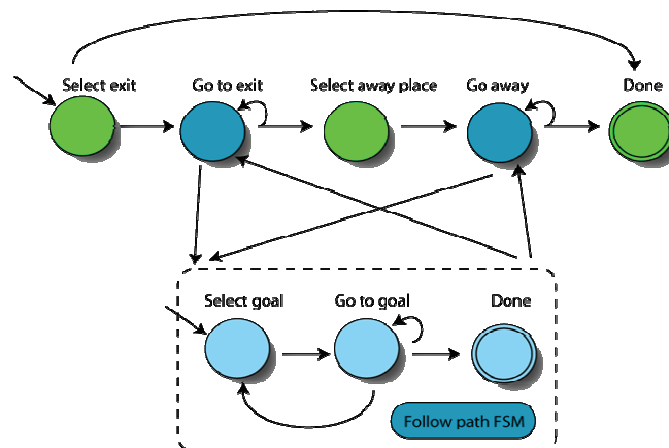


Figure 8.4: Stack of states describing more complex goals using sub-goals FSM.

As our system keep a stack of states to be executed, it is possible to create full sequence of event. The listing below illustrates the creation of tasks queues. In addition, these queues can serve for storing past events that we can backtrack for simulating short-term memory. This is possible by keeping a pool of past states in memory. The Appendix H present additional Metatables handling more advanced behaviors.

Listing Virtual Human State Sequence:

```
-----
-- go to a place sequence
-----
Goal_GoPlace = {}

Goal_GoPlace["Name"] = "Goal_GoPlace"

Goal_GoPlace["Enter"] = function(character)
    character:attachTask("SimpleTask",0)
end

Goal_GoPlace["Execute"] = function(character)
    -- find a pt in the node place
    local posDest = findLocation(character,Place[character:getPlace()])
    goToPlace(character, posDest )
    -- clear any remaining subgoal
    character:getFSM():clearSubGoal()
    character:getFSM():addSubGoal(State_PathBehaviorActivateGoal)
    -- here the sub HFSM force the character to reach the place
    -- at arrival, play some idle animation
    character:getFSM():addSubGoal(State_IdleStart)

    -- run the event (first state in queue)
    character:getFSM():popSubGoal()
end

Goal_GoPlace["Exit"] = function(character)
    -- nop
end
```

8.4 Variety

To improve the virtual humans' believability, every agent should have a distinct behavior. To increase the variety, our solution consists to use a bank of animation simulating the different behaviors. Then, we add noise heuristic to our motor functions in both the selection and the execution of every animation. The system select animations from a set, change their duration and blend different actions. This provides the feeling that every animation is unique. The listing below describe the randomness of animation selection

Listing Animation Variety:

```
--Animation slot: emotion time and a list of accurate keyframe animations
<listAction>negative 3 protest2 1.0 whistle1 4.0 whist2 4.0</listAction>

State_IdleNegative["Execute"] = function(character)
    if character:isActionNearCompletion(
        ACTION_NEGATIVE, ACTION_BLENDING_COMPLETION) then
        character:doAction(ACTION_NEGATIVE, false, 0.5 + math.random())
    end
end
```

8.5 Authoring

To author the agent's behaviors, designers need to rely on dedicated tools that provide the abilities to generate FSM and animation banks in an intuitive and high-level representation. The content creation pipeline combined both offline and online tools as illustrated in Figure 6.21. The next sections will introduce the tools developed for creating, editing, and analyze the run-time behaviors of virtual humans.

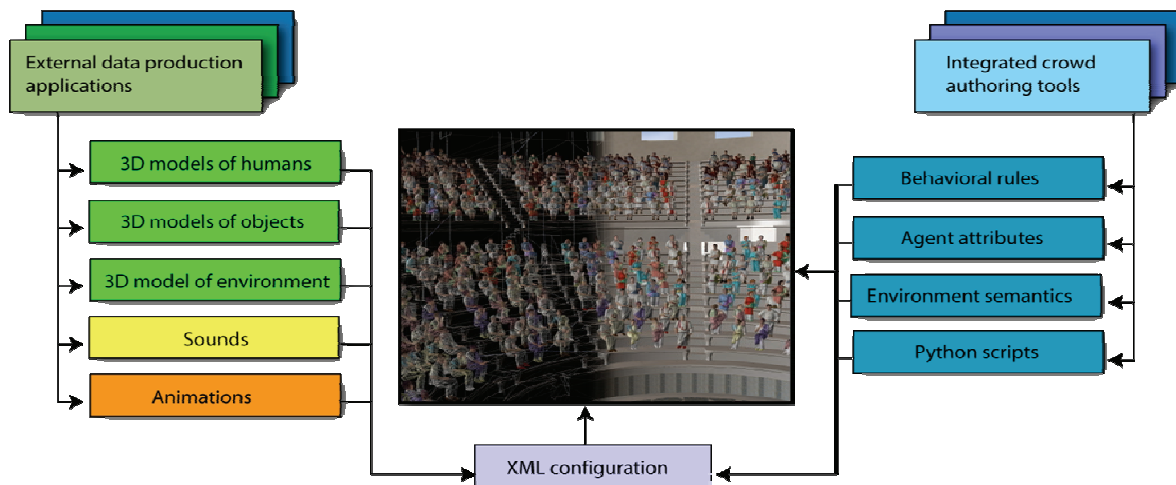


Figure 8.5: Virtual humans and behaviors authoring pipeline.

8.5.1 Offline Pipeline

With today's system complexity, developing a library alone is not enough. A dedicated effort for building tools that will unleash the library's potential is also necessary. Coding thousands of states by hand can be a painful process. Thus, allowing the generation of the FuSMs in a graphical environment can provide an important productivity boost. The Figure 8.6 shows a customized tool that has been developed derive from the work of [Darovsky2005]. It is used for setting state transitions, which then exports this data directly to their associated scripts. Different open-source initiatives exist, like the one from [Gill2004], [Jacobs2005] or [McNaughton2006] that could serve as a good point of entry for interested readers.

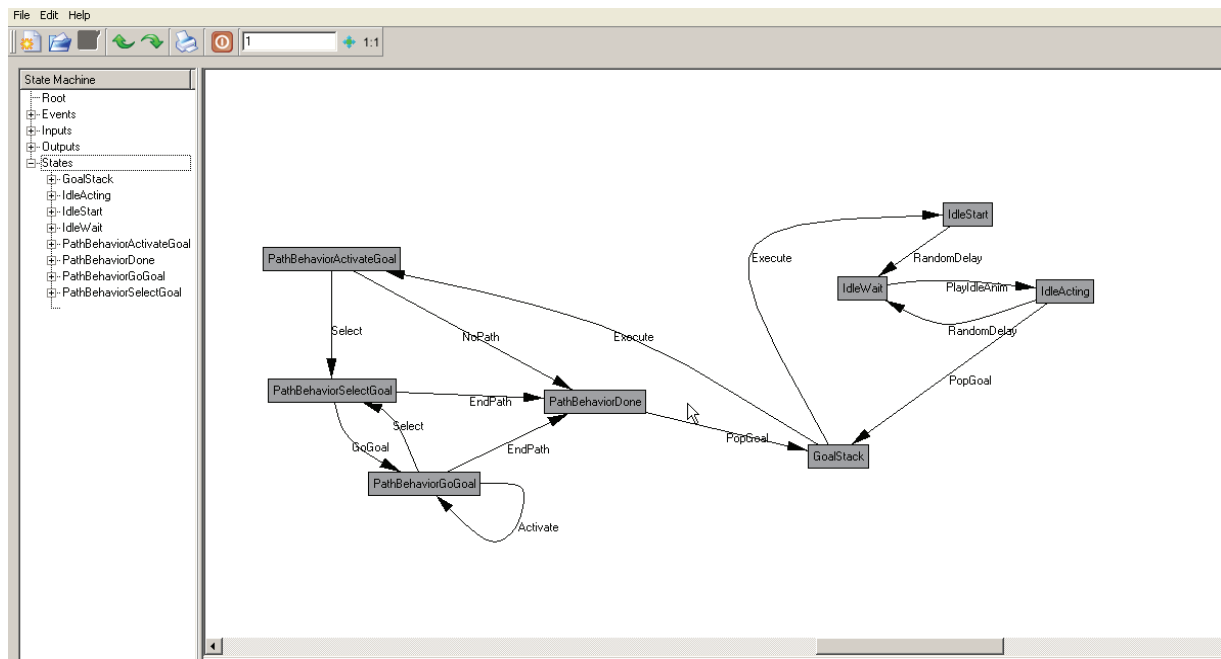


Figure 8.6: Designer Tools for creating the FuSM.

In the FuSM editor shown in Figure 8.6, the core idea is to store this information into XML configuration files. Then, a compiler generates the different FuSMs within Lua Metatables. Since using XML code within the intermediary files keep separate the authoring tool from the engine data structures. Figure 8.7 depicts this pipeline, showing that the runtime engine will transpose the Lua scripts into Lua byte code, which will in turn be used by the AI engine at runtime.

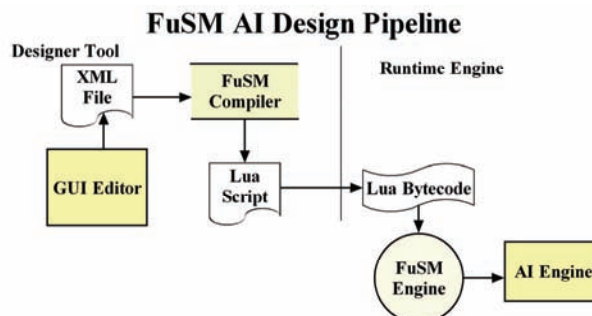


Figure 8.7: Pipeline for generating the different FuSM from the GUI editor to the runtime engine.

8.5.2 XML Configuration

The Lua metatable are uploading in our system during the initialization phase. The next listing illustrates XML tags specifying which Lua Scripts need to be executed.

Listing Lua Script XML Definition:

```
<vhdLuaScriptProperty name="motivation">
  <strScriptPath> ./data/lua/motivation.lua </strScriptPath>
</vhdLuaScriptProperty>

<vhdLuaScriptProperty name="HFSM">
  <strScriptPath> ./data/lua/fsm.lua </strScriptPath>
</vhdLuaScriptProperty>
```


8.5.3 Runtime Information

Developing 3DRTS, featuring hundreds or more dynamic entities (like crowd simulations) is a difficult process. To construct believable worlds, designers need to interact directly with the virtual environments. Seeing general action from the on-screen characters is not enough, they also need to modify the simulation parameters at the entity level during runtime. The idea is to generate dynamic GUIs that can reflect the current data set based on the designers needs. Figure 8.8 shows an example of such a GUI that provides visual debugging information for the current navigation graph used for path planning requests. It also features a set of widgets displaying feedback information for controlling the simulation. Users can interact directly with the system, triggering events or dynamically changing system parameters.

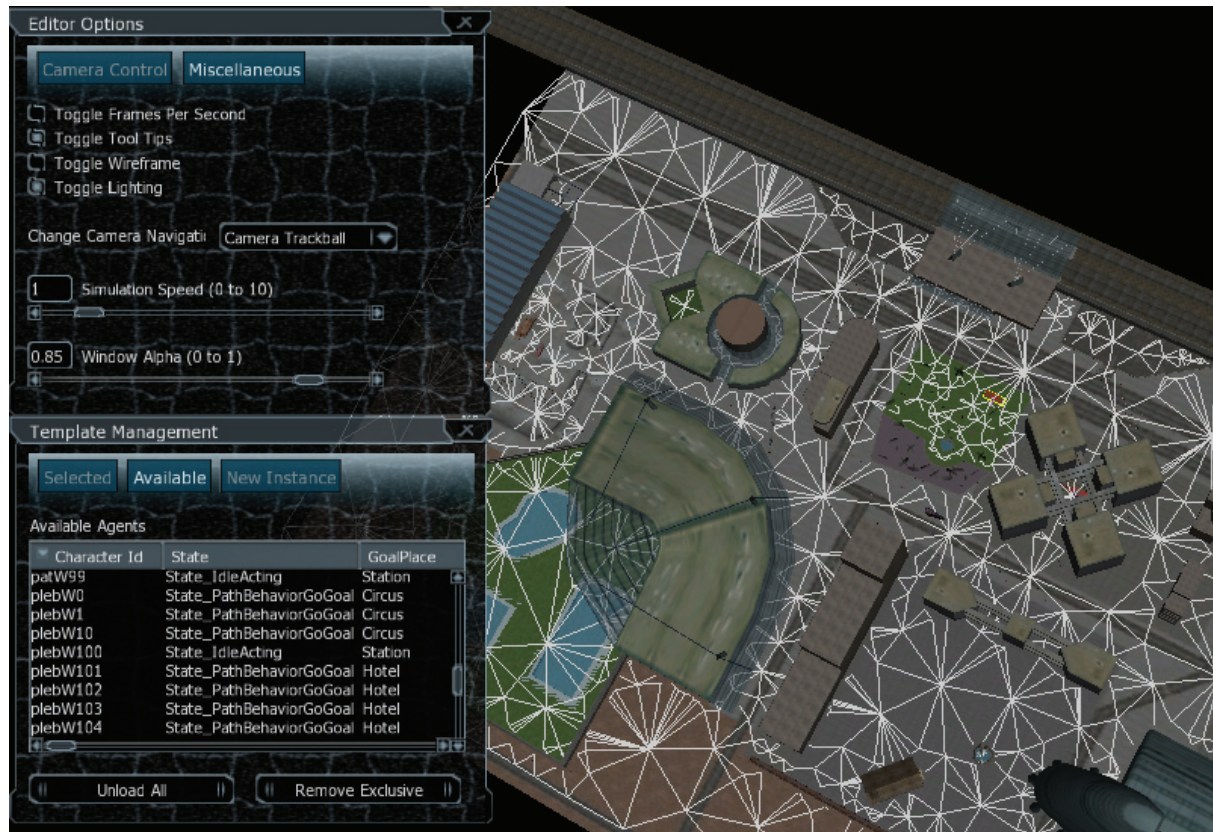


Figure 8.8: Real-time GUIs for controlling the entities state evolutions.

8.6 Performance

One common issue that somewhat plagues data driven systems is that of performance, since executing scripted routines is almost always slower computationally than straight code. All data driven engines must avoid unnecessary state updates, by scheduling among priorities, as well as limiting the work that is being done at the script level. Heavy floating-point operations or large searching type actions have to remain at the code level, while the scripts will usually only set parameters where appropriate. Keeping the scripts limited in this way allows the benefits of data driving your system from being outweighed by the overhead.

In the system described here, information stored within C++ objects is bound to the scripting interfaces on demand. One problematic area is with task classification. Since the engine does not have control at the level of task complexity or duration, designers need the ability to provide hints to the system like those described in the next listing. A custom profiler may be necessary for collecting information about each task, like its resource usage over time. Depending on those profiled results, tasks can be assigned to the appropriate classification group. For instance, tasks with small variance in performance are good candidates for constant, frame rate based execution (using the maximum time duration necessary to allow for the largest time based category). The

different level of details (LOD) for each task's management can benefit from predicates that will try to estimate the cost of each LOD based on heuristic functions.

Listing: Tasks classifications may differ among their LOD.

```
State_DoAction["Enter"] = function(character)
    -- hints for classifying tasks
    character.lod[0] = MaxTimeTask(0.01)
    character.lod[1] = MaxTimeTask(0.002)
    -- constant time
    character.lod[ATI-R2VB] = ConstantTimeTask(0.0001)
end
```

8.6.1 Tasks Ordering

Managing and ordering all the different tasks comes at a cost. In this system, scheduling takes the form of a priority-linked list based on time. For reducing the complexity, we break each task into small chunks. Each chunk will be executed in order, providing a constant time search function. The rescheduling is an $O(m)$ operation, where m is the number of tasks within a chunk. If the chunks are well defined, rescheduling will not occur very often.

8.6.2 Level of Detail (LOD)

Similar to graphics engines, which need to render multiple objects within the capacity of the current processor's power, the AI has to be able to provide different level of abstraction for controlling its entities. When an entity is close to the player, we expect very smart, rich behaviors. However, off-screen and distant entities do not require the same attention to details [Brockington2002, Robbins2003].

Determining the LOD classification for any given task depends again on several factors, like overall resource usage. In a graphical LOD system, the distance to the camera may serve to select the current LOD. Sound and behavior LOD systems may require much more specialized alternate heuristics. An AI LOD might take into account the computational complexity of performing the task. Animations within our system can be based on skeletal animation or use prerecorded gestures. In the former case, the computation would be based on the number of bones within the animation, whereas the latter would be a constant time task. Thus, the system will be able to provide more accurate dispatching of available tasks based on their current LOD as show in the listing above.

For ordering workflow, tasks need to be separated into different categories. One category contains tasks that need updating on a frame-by-frame basis. Another category holds periodic tasks, and another keeps tasks that are to be executed only when processing time is available. Some typical use cases of this last category are related to virtual characters that populate the world but that do not directly interact with the player. Thus, if such a character is playing some idle animation and needs to update its behaviors, the task can easily be postponed for a few frames, without becoming too noticeable to the player. To some extent, some tasks can also use statistical information collected by the engine to be reassigned to a more appropriate category on the fly (notably for periodic tasks).

8.7 Conclusion

Our data-driven AI engine offer good flexibility traits, which provide the ability to adapt and create new agents' behavior dynamically. By taking advantages of Lua metatable, our architecture minimize the interpreted scripting layer overhead. With the addition of managing LOD for agent's behaviors still allow to compute several thousand agents with interactive frame rate. This is generally sufficient for most 3DRTS, which benefit more from flexibility and extensibility than pure RAW performance. In the results sections, different use-case relying on this architecture will be investigated.

Chapter 9

Authoring

The best combination of software and libraries is meaningless, if it does not come with set of authoring tools to unlock the potential of the underlined system architecture. Developing core technologies is one-step in the development process. The development of tools is the second step, which is often neglected. In this chapter, we will briefly introduce the tools commonly find in 3DRTS frameworks and we will illustrate the tools specifically developed within the scope of this thesis.

9.1 Unlocking System Potential

Developing components or libraries that form 3DRTS systems do not suffice. Dedicated efforts have to be made on promoting tools and editors that allow controlling and creating simulations more easily, especially for team members with limited technical knowledge [Walker2006]. Being able to provide tools in which artists and designers can directly interact with the system is the key element to unlock the potential of the inner algorithms. For instance, [Wihlidal2006] emphasizes on the importance of creating efficient toolsets. He presents a comprehensive series of articles that illustrate how authoring tools utilities significantly improve 3DRTS developments productivity. Whilidal adopt C# and the .NET 2.0 framework, due to its rapid tools development traits and easy interoperation with 3DRTS engines. In effect, the interoperability between C# and C++ together with the high modularity of the .NET framework improve the production of robust tools [Maughan2006, Novaes2006].

Modern systems architectures for 3DRTS are equipped with some form of toolsets. For instance, the Figure 9.1 displays some tools use with the Unreal Technology [Epic-Unreal]. The left image represents the Unreal Kismet visual scripting system. This editor allows interconnecting simulations elements within a flowchart. Designers can build their own scripts without any programming involved. These free programmers to analyze scripts developed by designers. The right image shows the Unreal scripting console, which can be used at run-time for executing scripts on the fly, improving the fine-tuning of applications.

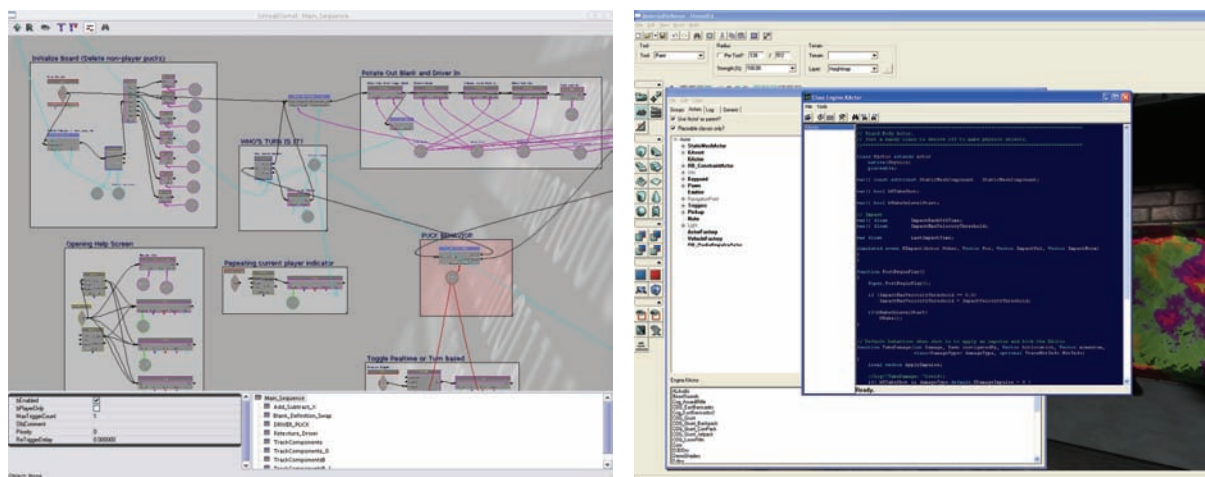


Figure 9.1: On the left part the Unreal Kismet Editor. On the right, the Unreal scripting console.

9.2 Object-Editing Toolkit

Object-editing toolkits are dedicated software that allows controlling and editing ISOs with a WYSIWYG editor. Many 3DRTS developers are developing object-editing toolkit for manipulating ISOs. The Figure 9.2 illustrates two different object-editing toolkits that have been made available for the mod community.

As we can observe, these editors provide feedbacks on the ISOs, allowing modifying dynamically their behaviors.

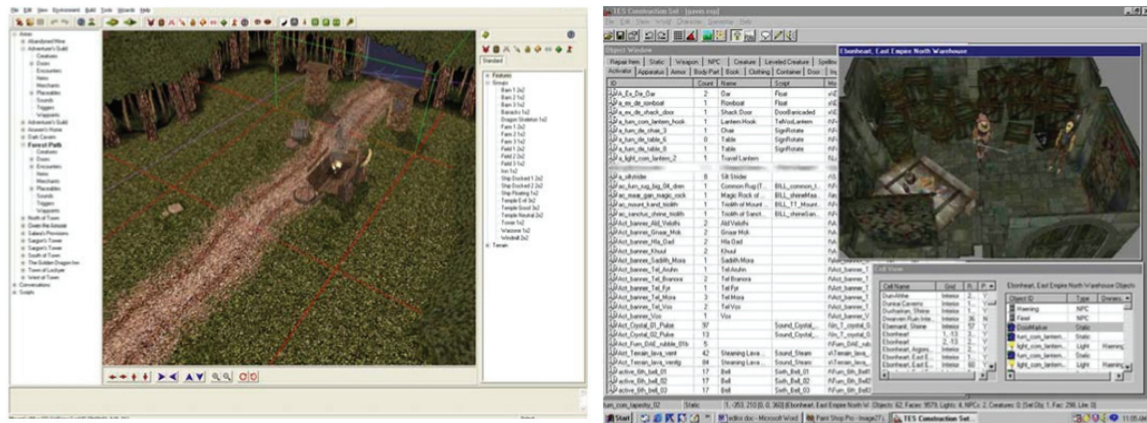


Figure 9.2: On the left, the Neverwinter Nights toolkit, and the right the Elder Scroll III: Morrowind toolkit.

In our system, the object-editing toolkit is not as complete as the one presented above, but still provides the set of functionalities required for controlling and modifying dynamically ISOs. In this occurrence, ISOs can be controlled by scripts. In effect, these scripts allow placing and setting objects properties. Many commercial games and VR simulations rely entirely on scripts. In fact, EA development teams are using intensively Lua as a scripting language for controlling ISOs. For instance, Radon Labs and his open-source game engine, “The Nebula Device” [Radon], uses tools that automatically generate Tcl/Tk scripts, which are interpreted by the engine.

In our case, the data manipulation is done using different widgets dedicated for each aspect of the simulation. As shown in Figure 9.3, we rely on an open-source editor based on OpenSceneGraph. These tools allow editing and manipulating graphical objects within the scene graph such as providing hints to the rendering pipeline. These modifications can be used at runtime and the resulting modifications are directly store into XML files.

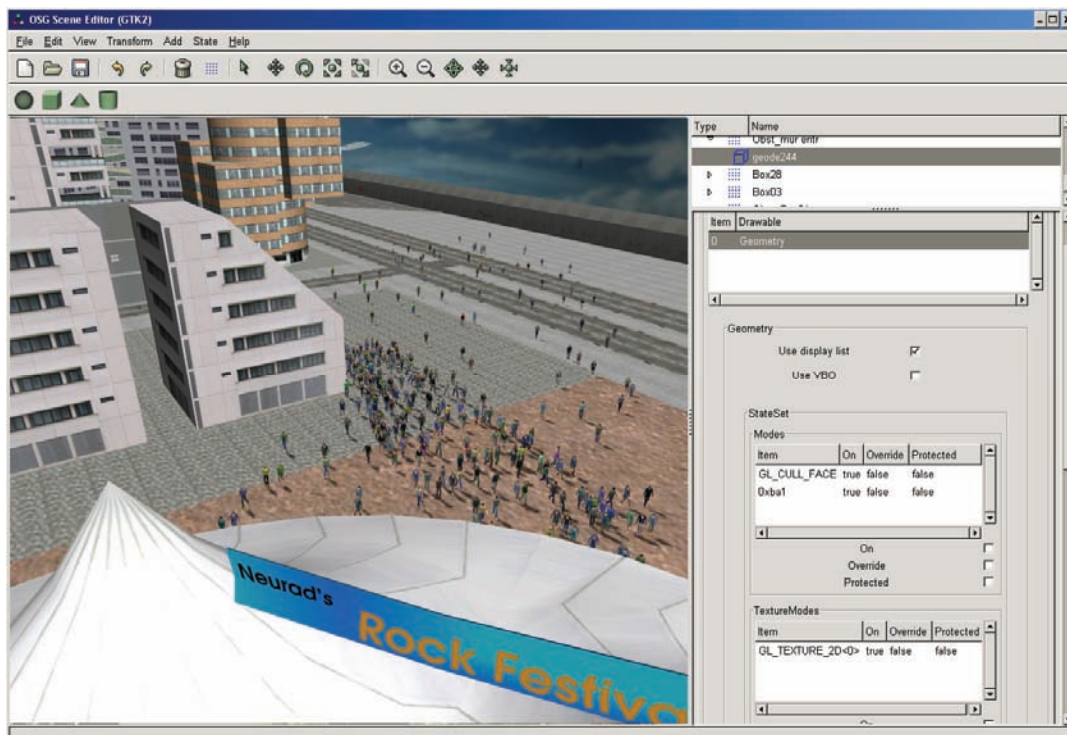


Figure 9.3: OSGEdit allow to edit graphical object properties and to analyze the internal scene graph.

9.3 Assets Management

As Modern 3DRTS developments move toward creating richer, more detailed worlds than ever before, the necessity to manage assets efficiently is also increasing. Digital assets management represents techniques that provide potential solutions that can be employed to tackle these issues. The different stages of simulations content pipelines generally start from a 3D modeling package up to the final engine data structure. Between these two steps, different intermediate tools are responsible for pre-processing and packing data structures. [Marselas2003] showcases the methodology used during the development of the video games Age of Empire II. He describes that the development team considered creating their own tools for managing the game assets because of the lack of good front-end solutions from existing assets management software. He explains that such tools are very capable of collecting and storing massive-scale contents but do not provide enough flexibility to facilitate the manipulation from the original files up to the run-time data structures. [Chatelaine2003] describes how to enable data driven tuning using existing tools such as Microsoft Excel. Notably, he illustrates how to generate XML files by adopting the mechanisms provided by [Infopath]. The problematic of content creation pipelines was also covered by [Carter2004]. He describes all the steps involved from coverage of version control systems to assets manipulation and exporters. He provides some generic examples that are commonly found in many 3DRTS developments. [Sweeney2005] presents the new challenges faced for developing up-coming 3DRTS from both a technology and content creation standpoints. He describes the needs to develop tools for empowering artists to create great content, directly inside the design engine, without needing additional help from programmers. He address that artists-oriented tools are the keys to harnessing and managing the power of the next generation of hardware. [Green-ART2005] describes the classical approach of art pipelines and presents his concerns about their abilities to handle the growing production needs. He is addressing these problematic by presenting alternatives that are more efficient. The Figure 9.4 depicts the different layers in the content creation pipeline.



Figure 9.4: The hierarchy for content management pipeline.

9.4 Content Creation Pipeline

The content creation for 3DRTS repose on several steps and involves different team members from artists to programmers. In much systems architecture, developers limit themselves to export data from DCC tools directly to their own run-time files formats. In our framework and similar to other frameworks such as RenderWare [Criterion] or Unreal [Epic-Unreal], we extend the pipeline to provide a middle layer represented by metadata. The Figure 9.5 describes the content creation pipeline use within the system. This pipeline is clearly separated into two main categories: the platform independent and high-level scene representation and the run-time and platform specific data. On top of this hierarchy are defined set of plug-ins that extend the capabilities of DCC tools. Notably, we use the Bone Pro [Digimation-BonePro] plug-in for 3D Studio Max [Discreet] that facilitate the creation of virtual humans' skeletal deformations and accelerate the rig and animation of detailed characters. From the DCC tools, artists model directly the 3D models within real-time boundaries and export them into an XML file or as OpenSceneGraph native file format for quick previzualisation of contents. After the contents exportation, developers are free from any DCC tools but are also

able to merge multiple contents created through different tools, by applying the intermediate metadata files format. Using the object-editing tool describe in Figure 9.3, the scene organization can be changed and visualized outside the main application. For run-time performance, it is important to adapt the assets upon the hardware capabilities. This is made possible by converting the metadata into a binary format. For graphical elements, we use [osgConv]. This tool is able to optimize 3D assets for run-time usage, adapting the complexity by creating geometrical LODs and textures compressions whenever the end-platform support it, such as DXTn or 3Dc algorithms. The resulting archive containing the 3D scene information will then be stored using the LZO compression algorithm to minimize the data transfer even so it requires real-time decompression of data. Experiments have shown that the overall loading time can be reduced by compressing the data as the ratio of CPU performance and data storage bandwidths increase with time [Bilas-ISL2001]. Finally, the run-time application will describe in an XML file, which archive need to be loaded using the LZO compression algorithm [Oberhumer].

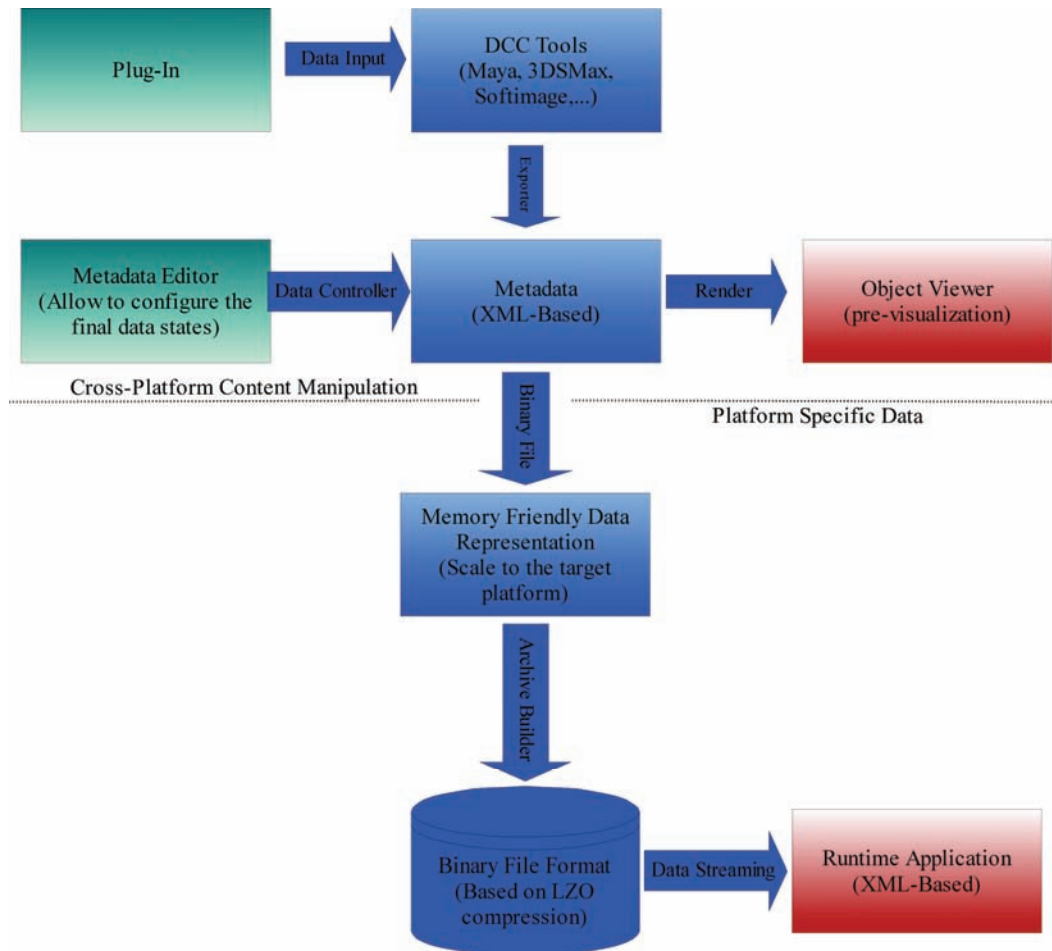


Figure 9.5: Content creation pipeline.

9.5 Exporters

Over the years, our designers have created many assets including H-ANIM WRK animations files and virtual humans' skeletons. As resources for creating and adapting content to new technologies were restrained, we opted for a consistent approach, which rely to convert existing assets. To do so, we have created a series of python scripts, which accept as an input a .tpo and .wrk files describing both the virtual humans skeletons and animations files and output the corresponding files used by our crowd engine. These tools give us an instant library of animations and allow creating dynamically walking and running animations files based on the work of Glardon [Glardon2004].

9.6 GUIs for Interactivity

For interacting with the crowds and obtaining information about the status of each character, such as their current states or goals, specific widgets have been developed using the CEGUI library [Turner]. This allows creating dynamically 2D widgets on top of the 3D window. In our system, the GUIs are used in two ways. This provides statistical information retrieved from the underlined AI engine and they allow affecting events to group of individuals. Sometimes, end-users want to interact directly with selected virtual humans [Thalmann2004]. To do so, we have extended the crowd brush concept from [Ulicny-Brush2004] allowing to define brush and related events in a data-driven fashion. Therefore, we can re-assign the inputs device context at run-time. The Figure 9.6 highlights the two main interactivity GUIs.

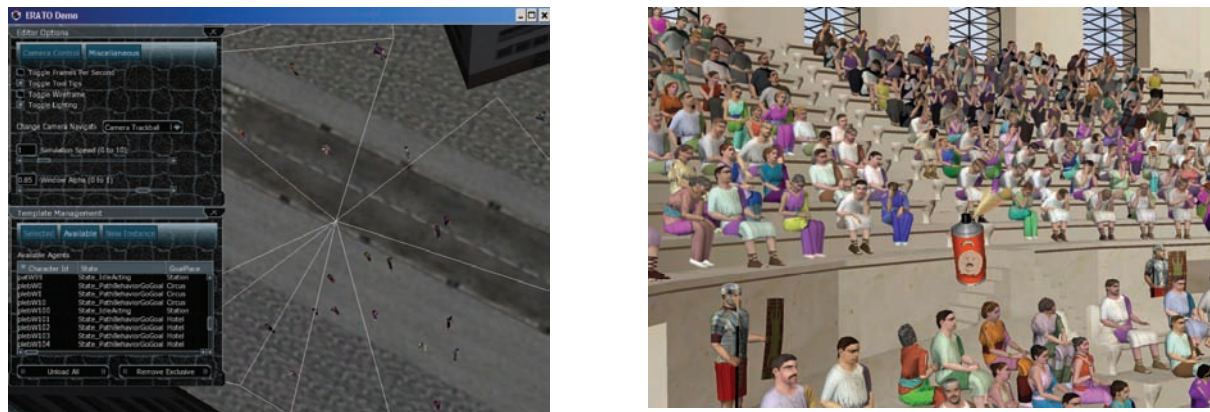


Figure 9.6: On the left, set of widgets for interacting with virtual humans, on the right crowd brush to dynamically affect emotion to virtual characters.

9.7 Input Device (Gamepad)

To control virtual characters in a convenient way, a specific module handling gamepad as an input device was developed. It relies on the DirectInput library [DirectX] and offers the possibility to re-assign inputs on the fly. This is useful as it allow running the same simulation with different inputs devices by simply re-affecting the events. Additionally, our module support force feedback effects that can be generated using the editor FEdit that come packed with the DirectX SDK (see Figure 9.7).

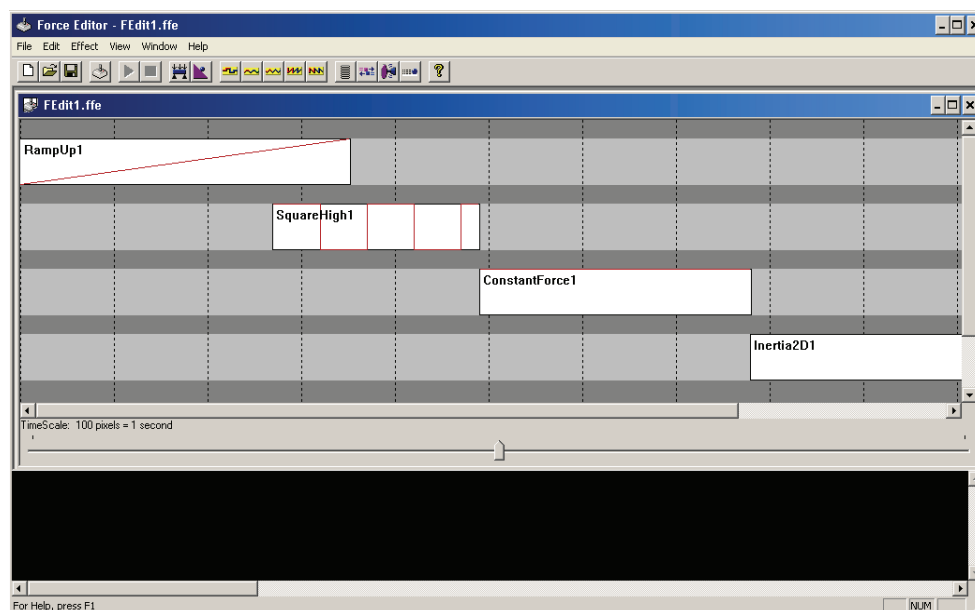


Figure 9.7: The FEdit Editor for creating force feedbacks effects files (.ffe).

This allows taking control of individual characters using a third-person interaction paradigm. Third-person view of the character is a common interaction with computer games: a player is controlling a character that can be seen on the screen performing actions according to commands given by keyboard, mouse, joystick, or any combination thereof. The Figure 9.8 describe one navigation mode, which rely on a procedural walk engine [Boulic2004] to drive the character animation based on user inputs coming from a gamepad.

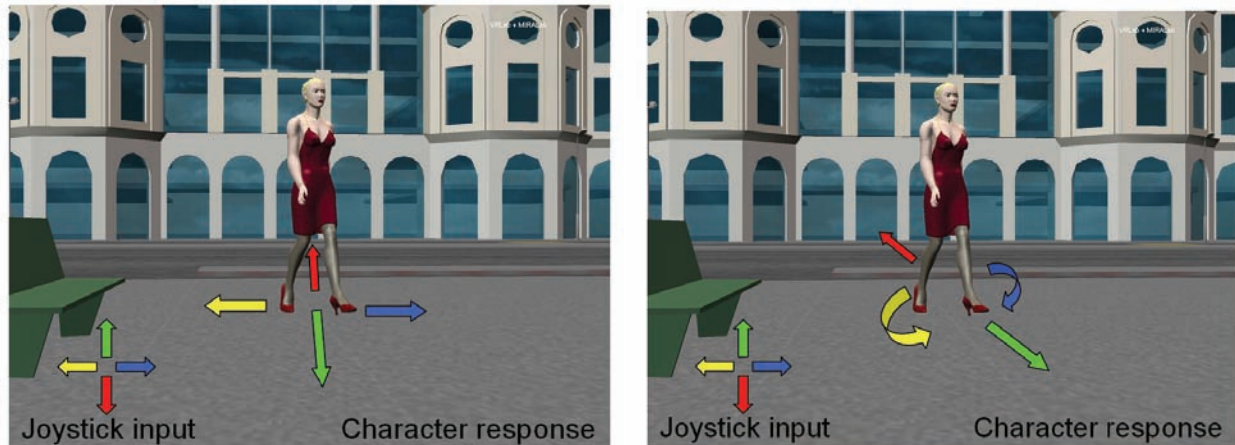


Figure 9.8: The left analog stick move the character, the right analog stick allows turning the character.

A gamepad allows you to enter the input in several ways: it includes analog and digital joysticks, buttons, and a slider. In this example, we control the walking using two analog sticks. All the input and others channels that may be use to triggering actions are developed within a python script as described in the listing below:

Listing Input Device:

```
event = UserInputPtr(inputService.getUserInput())
while (True):
    if event.fLeftAxis or event.fRightAxis:
        hagentService.walkSetDesiredPositionAndDirection(
            avatar, destPosition, destDirection)
    elif event.bButtonA:
        hagentService.kfSetGlobalTranslationMode(
            avatar, 'animA', , "KEYFRAME_RELATIVE_MODE")
    #... other input
    vhdYIELD #wait until next update
```

Examples that are more complete can be found in Appendix G. Another alternative would be to take advantages of the embedded python interpreter to adopt COTS modules such as PyGame [Schinners2005] for handling input devices.

9.8 VHD++ Technology Map

The creation of 3DRTS requires combining many tools that can handle the different specific aspects of the development. Some tools will handle the code generation why other tools with give the ability to create simulation content such as textures or 3D meshes. On top of that, many tools are used to manage all the assets from files converters to source versioning or documentation generators. The Figure 9.9 illustrates some of the tools and applications used by our framework.

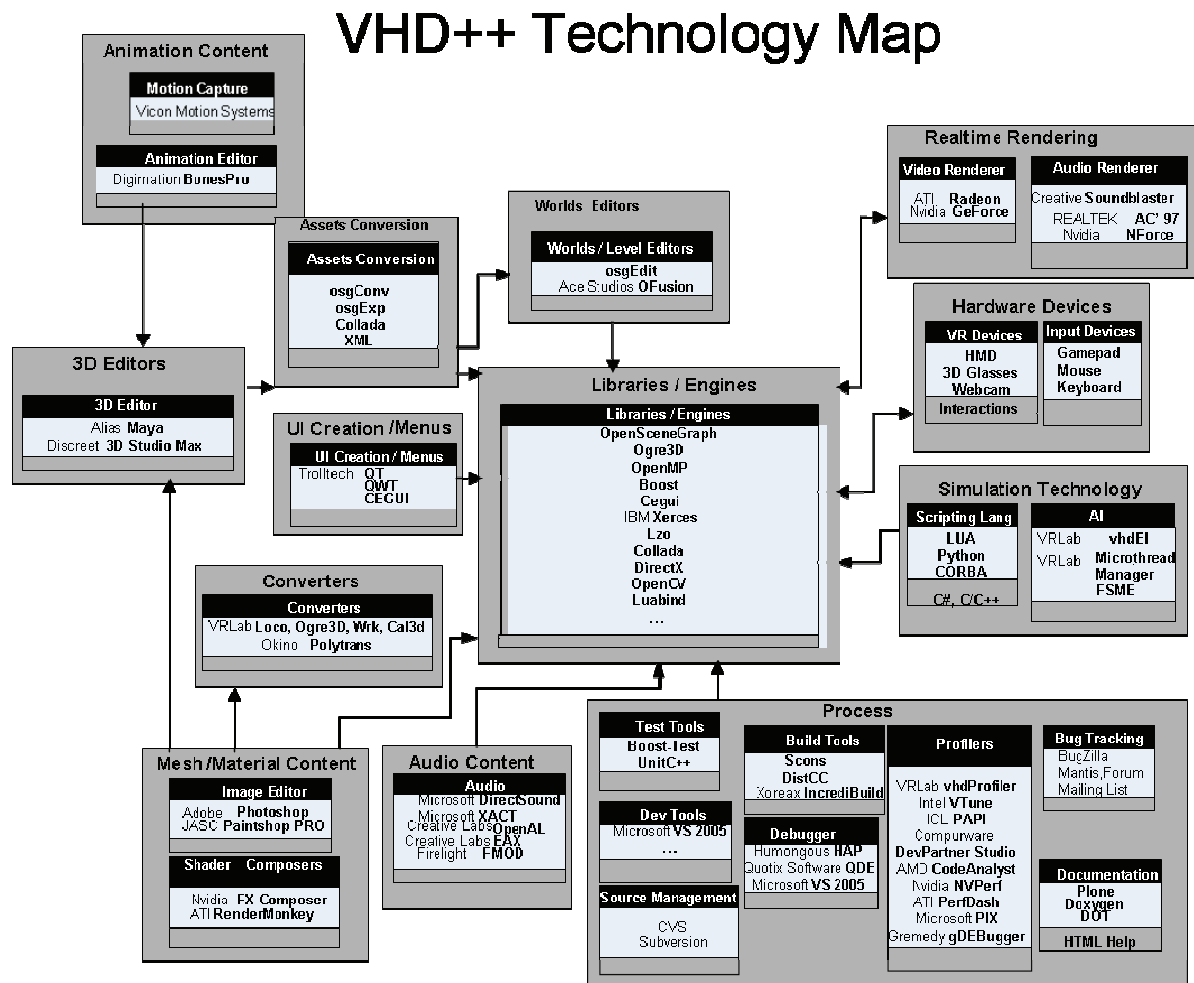


Figure 9.9: The VHD++ technology map.

9.9 Profiling

Developing a concurrent architecture for 3DRTS rely on the proper load balancing between the different components involved [Buck2004]. One difficulty is to discover the relative computation costs and system bottlenecks [Maquire1999, Marselas2000, Asaduzzaman2003, Kiel2006, Trebilco2006, Zarge2006]. Different situations are happening at the same time. We need to analyze if the problematic come from routines that are heavily being called or does it comes from accessing shared resources. Systems developers need to analyze the system behaviors with cautious. The system has to present a set of functionalities for controlling the simulation workflow as it may change for every single situation. Assuming the same platform can be used for a wide variety of simulations, their internal requirements will adjust the global behavior of resources usage. For instance, a crowd simulation requires important processing power for graphics and AI management, while an augmented reality simulation dedicate more energy for tracking camera images.

9.9.1 Commercial Profiling Tools

Different commercial tools have been develop for helping developers to profile 3DRTS [AMD-CA, Intel-VTune, Metrowerk-Analysis, NVIDIA-NVPerfKit, SN-Systems, Lemarié2002, Domine-GPUJ2004, PAPI2004, Aguaviva2005, Performance_Tools_Lab2005, Kiel-Perf2006]. For concurrent systems, Intel has extended the VTune application with specific plug-ins, which helps discovering simulations behaviors within a multithreaded environment. The application will describe potential race conditions as well as describing wasted

CPU cycles, such as when a thread is waiting for accessing shared resources. All these information is a great help for discovering the system limitations. The performance analysis goes beyond evaluating the execution speed and memory footprint of the different subsystems, it includes measurements for analyzing the ways the different components interact. One of the most useful and profiling toolset is VTune. This software provides performance counters that collect, analyze, and present data flow in meaningful diagrams. VTune works as a sampling profiler, which mean it will wake up every few delta time and will produce statistics information for each process executed during that time. One benefit of VTune over alternatives software, is that it does not rely on specific code generation and do not reduce the performance by a significant factor. However, sometimes it may be difficult to extract precise information concerning the profiled program from the other processes. As a note side, one difficulty related to profilers, is that they may reduce processing throughput, and by consequences, some bottlenecks related to others elements such as GPUs may disappear. Also due to the multi-threading nature of most OS, other programs can interfere with the collections, obliging to use several profilers for discovering unpredictable behaviors. In general, it is always better to restraint the profiling on smaller sections of code to improve the accuracy. However, these profiling sessions provide information on a global scale. It is difficult to configure such tools to profile fine-grain and specific code sections with ease. Moreover, tools like VTune work on a limited number of platforms and may not be suited to run on embedded platforms or for full-screen modes [Olsen2000, Rabin2000, Hamm2004]. This leads to the conclusion that customized profilers have to be designed for those specific needs. In our system, a dedicated profiler SDK was developed in that regard. The next sections introduce our inner profiler from the code usage and its functionalities as well as its controller GUIs and files exporters.

9.9.2 Custom in-Simulation Profiler

The principles of the present profiler API are to provide an interactive profiler that intensively instruments the code. The code is divided into zones by programmers, which use a set of macros that facilitate to compile the system with or without profiling support. These macro measures the times spent within the current scope. When a Zone is calling another Zone, two separate times are stored: the time spent in the entire zone and this time minus the embedded zonetime. This clear separation allows distinguishing more easily, which are the intensive processing zones, by going down the hierarchy. The developers will first analyze which high-level components are too processing intensive and, then will be able to track the set of routines that requires deeper analysis. The Table 9.1 depicts the functionalities available by our profiler.

Table 9.1: In-Simulation profiler functionalities comparison.

Profiler	Blows' Profiler	IProf	vhdProfiler
<i>Self Time</i>	Yes	Yes	Yes
<i>Hierarchical Time</i>	Yes	Yes	Yes
<i>Average</i>	Yes	Yes	Yes
<i>Variance</i>	Yes	Yes	Yes
<i>Call Graph</i>	Yes	Yes	Yes
<i>Feature Map</i>	Yes	No	No
<i>Recursive</i>	No	Partial	Yes
<i>Thread-Safe</i>	No	No	Yes
<i>Export File</i>	No	No	Yes

At run-time, the diagnostics information is divided into slots. The profiler records the total time spent on the previous slot as well as storing the zone information. Like the profilers from [Iprof, Evertt2001, Hjelstrom2002, Blow2003], the current profiler keep the hierarchical time representing the overall time spent in that zone as well as the self-time which do not include the time spent in children zones. In addition, the profiler will keep the average time spent in the zone as well as its variance. The variance is one of the most important elements as it may indicate that some algorithms have duration. Such algorithms may include A* algorithms as used in many navigation graph methods. As the profiler is keeping a list of zones callers, developers can observe if the duration of a zone is related to a specific zone caller. Similar to the [Iprof] profiler, the system also support recursive routines including call depths but overcome its limits when reporting the recursive data information. Nevertheless, the main contribution of this profiler is to feature thread-safety abilities.

9.9.3 Profiler User Interfaces

Our profiler can export diagnostics information that can be loaded in tools such as Microsoft Excel. However, many times, developers expect to observe and analyze the information at run-time. Therefore, we have developed a series of GUIs that highlight the system performance using the advanced scientific widgets offered by [Qwt] (see Figure 9.10). The left image capture the instant frequency or variance of the measured zones, while the right image present an historic of time spent at every simulation frame.

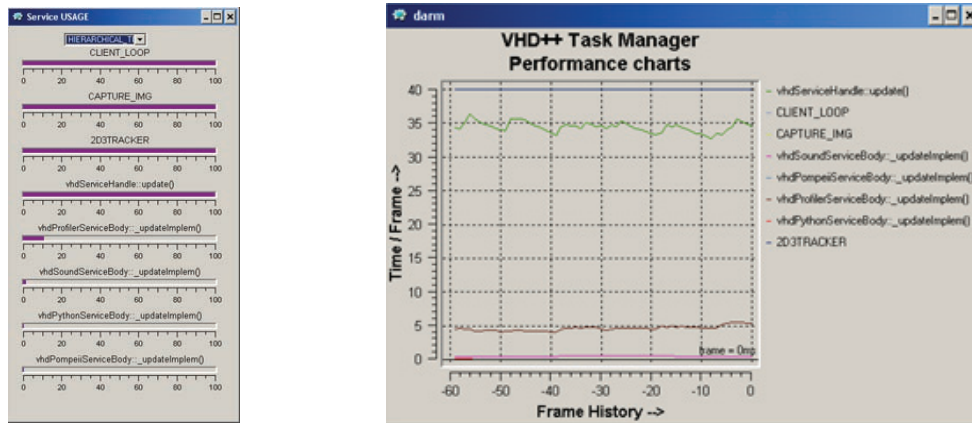


Figure 9.10: Diagnostic and profiling widgets.

9.9.4 Profiler API

The design principles behind the profiler API concern the usability of the library. It has to offer the ability to compile source code without profiling support avoiding polluting low-level libraries with additional dependencies. Generally, the profiler is not intended to run on too low-level routines, due to the timer precision limitations. Best candidates are components' update or zone with sufficient computation. It is inaccurate to profile few instructions or small loops as the precision and overheads prevent qualitative results. The library usage relies on very simple macros that are expanded accordingly to specific options that include thread-safety support and data sorting. Naturally, the profiler itself is taking few CPU cycles, whenever it is entering or leaving a profiled zone. If the zones are carefully chosen, avoiding intensively called zones, the overhead remains minimal (less than 1%). The listing below depicts how to declare profiled zones directly into code.

Listing vhdProfiling Macro:

```
void doRendering()
{
    //scope to be profiled, where "myRenderZone represent ZoneID
    vhdPROFILE_SCOPE("myProf"); //profiled scope with zoneid == "myProf"
}

int main()
{
    while(true)
    {
        vhdPROFILE_SCOPE("globalZone"); //scope to be profiled
        doRendering();
        vhdPROFILE_RESET; //proceed next slot
    }
}
```

9.9.5 Profiling and Performance issues

Many performance bottlenecks can be categorizes into groups [Marselas2003, Gee2005]. Theses groups allow analyzing the behavior of suspect code classify from executing dead code to inefficient code. Every major system will suffer from these side effects:

- *Inefficient code*: in large system, some code sections may not be efficient. If those sections require intensive computation, rewriting them is ideal for performance optimizations.
- *Processing too big datasets*: 3D environments are composed of many objects, and are indicators of performance issues. Thus, spatial algorithms allow computing elements that are within boundaries from the camera's position. If the system does not perform efficient culling or early removal of objects, the system may suffer of performance penalty by processing objects that will not influence directly the simulation. This is obvious for 3D graphical objects, which should not be performing when they are not visible at a given frame.
- *Inefficient memory usage*: poor performance can be the results of data-structures, which are not cache coherent, producing random access to main memory or allocating/freeing memory as well as data dependencies. Many data structures in performance-critical areas may benefit from representations that are more compact. They may fit nicely into cache lines, improving memory access.
- *Executing unused code*: over the course of a long development cycle, some functions may become deprecated but continue to be executed, spending unnecessary processing resources.
- *Bad usage of data structures and algorithms*: sometimes algorithms that perform well in certain situations may not scale for different usage. For instance, a data structure coming with some performance overheads may be acceptable on small datasets but may have a performance impact on bigger datasets due to cache coherency.
- *Special situations*: some algorithms such as spatial algorithms are directly dependent of the provided data. If such data do not allow splitting the environment into leaves, we may encounter the situation where the entire environment is defined in the higher branches of the hierarchy reducing the advantages of early objects removal. This may cause performance problems even so the algorithms by themselves are not responsible of the performance impact, because of incompatible or incoherent datasets. This happens in many occurrences, as most code is written base on assumptions on the data and specific boundaries.
- *Compilers*: many programmers do not learn the different compilers optimizations settings and stick with standard "release" default options. By analyzing the behavior and generated code from different compilers flags or C++ compilers, significant performance improvements can be obtained at no development costs [Gatlin2005b, West2005]. This is particularly critical for embedded systems, where the applications are running on fixed architectures allowing optimizing the code generation for this particular system.

9.9.6 Improving Performance using Scalability Tests-Units

Performance improvements can reach some limits when the system runs on a limited number of datasets and hardware. To investigate further the limitations and potentials performance improvements, it is important to be able to scale the system on different layers. This can be applied on the code itself by providing different implementations from a fast algorithm with low details to medium and high details implementations. This offer the abilities analyzing different code paths for the same functionalities (apart the loss precision). The second scalability factor rely on running the system on a wide range of hardware to analyze how the system perform with different set of CPUs, GPUs and memory. Finally, the last and more scalable element concerns the datasets. By changing some parameters such as screen resolutions, or using different models, gives records that help improving the system performance.

9.10 Conclusion

This chapter illustrates the need of efficient assets pipeline. We demonstrated some GUIs directly developed for our framework (see Appendix C for actual screen captures), which provide both feedbacks and interactions modes for the development team. Modern frameworks require dedicating more time and efforts for developing tools than focusing on core technologies. Even so many tools and GUIs were developed within the scope of this thesis; time did not allow providing a complete full feature pipeline for non-programmers. In effect, many tools and GUIs remain confined for programmers. The next chapter illustrate some results obtained using our framework (code and tools), which illustrate the system versatility in different 3DRTS gender from AR to VR simulations

Chapter 10

Case Studies

For confirming our architecture from both flexibility and efficiency standpoints, different simulations were developed. They illustrate how the system can be configured for a wide variety of simulations, including the interconnections with input devices such as HMD (head-mounted display), 3D trackers, or game pads (see Figure 10.1). This overview showcase different works using the system architecture, most of them are directly connected to EU projects, and were developed by multiple developers, representing a collaborative work. The first simulation to be analyzed is the EU Project JUST [JUST]. The goals of this project were to address the domain of training for non-professional health emergency operators, through the usage of VR based simulations. In this example, we will describe how specific components can be created handling custom VR devices. The second use case reconstructs a virtual orchestra where end-users play the role of the conductor using handheld devices and 3D trackers within a concurrent system. The third example is connected to the LIFEPLUS project, which intends to reconstruct old heritage site such as Pompeii in an augmented world. Here, we describe the flexibility of the platform for handling the particularity of augmented reality simulations, which features heavy computations for 3D real-time tracking as well as the management of scenarios using python microthreads. The fourth use case describes a simple demo that illustrate that the system is not bind to any particular component including the 3D renderer. If most of the simulation take advantages of the high-end performance of the OpenSceneGraph toolkits, this example show that the component can be interchange with other 3D renderer (here Ogre3D), without losing other functionalities such as H-ANIM animations. The fifth simulation describes briefly the simulation developed by Etienne de Sevin during his PhD thesis, which is built on top of the platform. In his work, he is taking advantages of cooperative multitasking operations through python microthreads for controlling and managing autonomous agents using prioritized emotions. The sixth simulation focus on the results obtained through the EU project [Erato], which target reproducing crowds within an Odeon in roman time. Finally, the last use case will highlight the potential extension of the platform for handling specific 3DRTS such as crowd simulations within an urban environment.

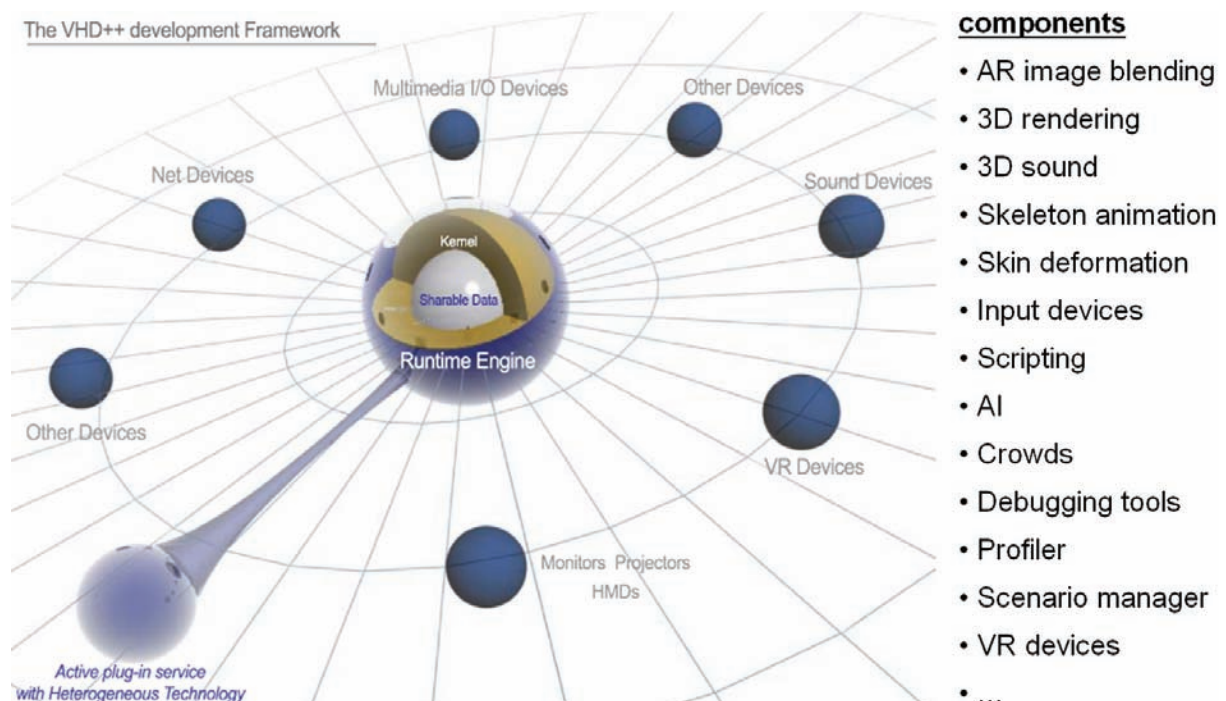


Figure 10.1: Spectrum of applications.

10.1 JUST

The JUST EU IST Project intends to create VR simulations use for learning best practices guidelines for training non-professional health emergency operators (see Figure 10.3). A User Group of recognized experts in emergency medicine training defined the training content. The simulation features a set of VR verifications tools that check the operator's capabilities to adopt the correct decision-making procedure. From the technological view, the VR challenge did not change much since [Sutherland1965] first experiments: trying to achieve sense of presence as an interface metaphor to a virtual world. Despite of the significant technological improvements over the last 40 years, the intrinsic limitations of the immersive hardware and the man-machine communication complexity are still there. As [Gobbetti1998] conclude: *"it is common to have misconceptions on what VR can and cannot do, and to have negative reactions when noticing that VR is not that real"*. With health emergencies training, we face the same major classes of problems. The interaction devices available do not allow for a large scene exploration and a direct manipulation simultaneously. The two are mutually exclusive because of the haptic feedback kinesthetic interfaces cumbersomeness. Finally, the medical users expect a realistic simulation, particularly in case of the human body appearance and behaviors. This of course conflicts with the real-time requirements of such interactive applications and thus must lead to certain trade-offs.



Figure 10.2: JUST emergencies simulations.

10.1.1 JUST VR System

As the JUST project is targeting civilian applications, so we had to face a challenging list of requirements (advanced VR system, easy to use, easy to maintain, extendible) and constraints (limits of VR hardware, limited budget, affordable PC platform). Finally, it was important that the high-end VR hardware components could be replaced by low-end ones and that the system is still fully functional. This gives the ability to have particular installations of the same system ranging from high-VR-end (e.g. stereo vision, surround sound, natural navigation) to low-multimedia-end (e.g. monitor, stereo speakers, mouse navigation). Being conscious of the limitations of the “direct immersion”, haptic feedback technological limits, and finally the above list of requirements and constraints, we propose a concept captured schematically in Figure 10.3.

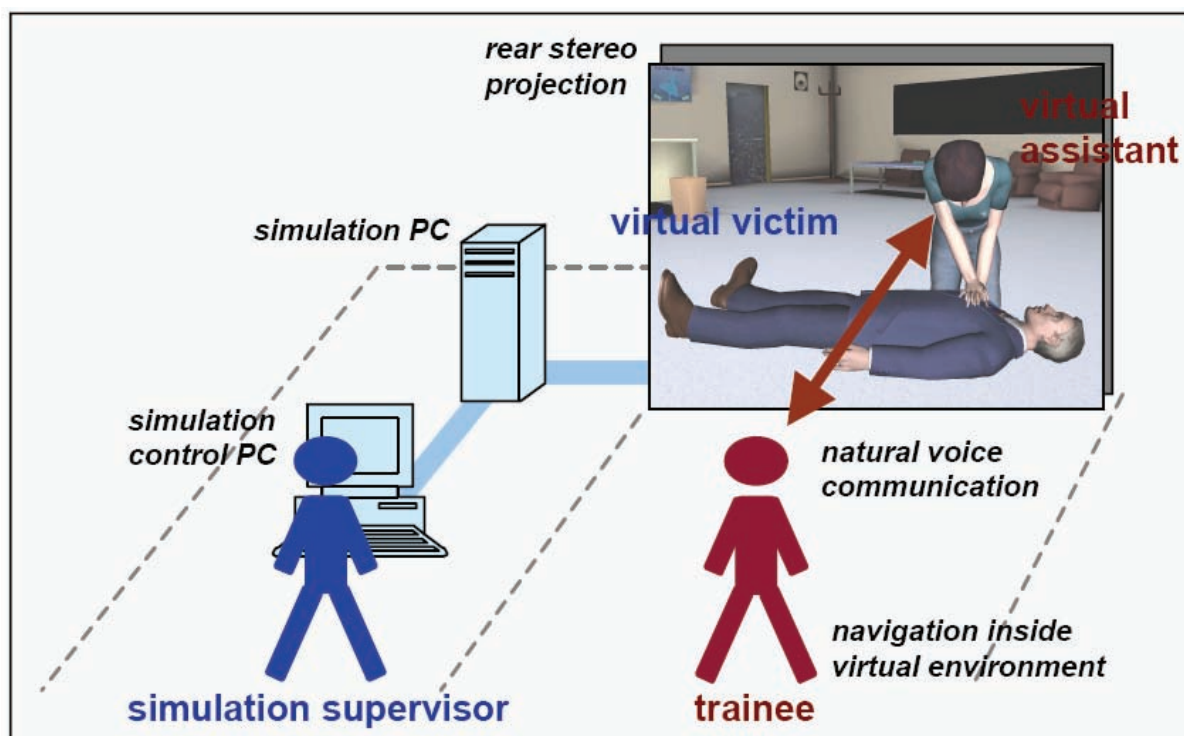


Figure 10.3: JUST VR system concept.

In short, during the interactive scenario the trainee faces a huge rear-projection screen displaying stereo images of the simulation. He is immersed in surround sound. In course of the simulation, he is able to navigate freely around the virtual environment to locate an emergency site. The intuitive navigation paradigm is based on a single magnetic tracker attached to the trainee’s head (see Figure 10.4). On the emergency site the trainee is interacting with his *Virtual Assistant* (VA) using natural voice commands and hearing respective replies and prompts from VA. The role of the trainee is to assess the situation and gives commands to the VA who is executing those showing proper manual skills. Simulation supervisor stays behind the scenes and has the following tasks: direction of the scenario path, “speech recognition” of the voice commands given by the trainee and triggering respective actions to be executed by VA (the supervisor can be regarded as the “ears of the VA”). This serves as putting pressure on the trainee by triggering of prompts spoken by VA or triggering of “disturbing” virtual events.

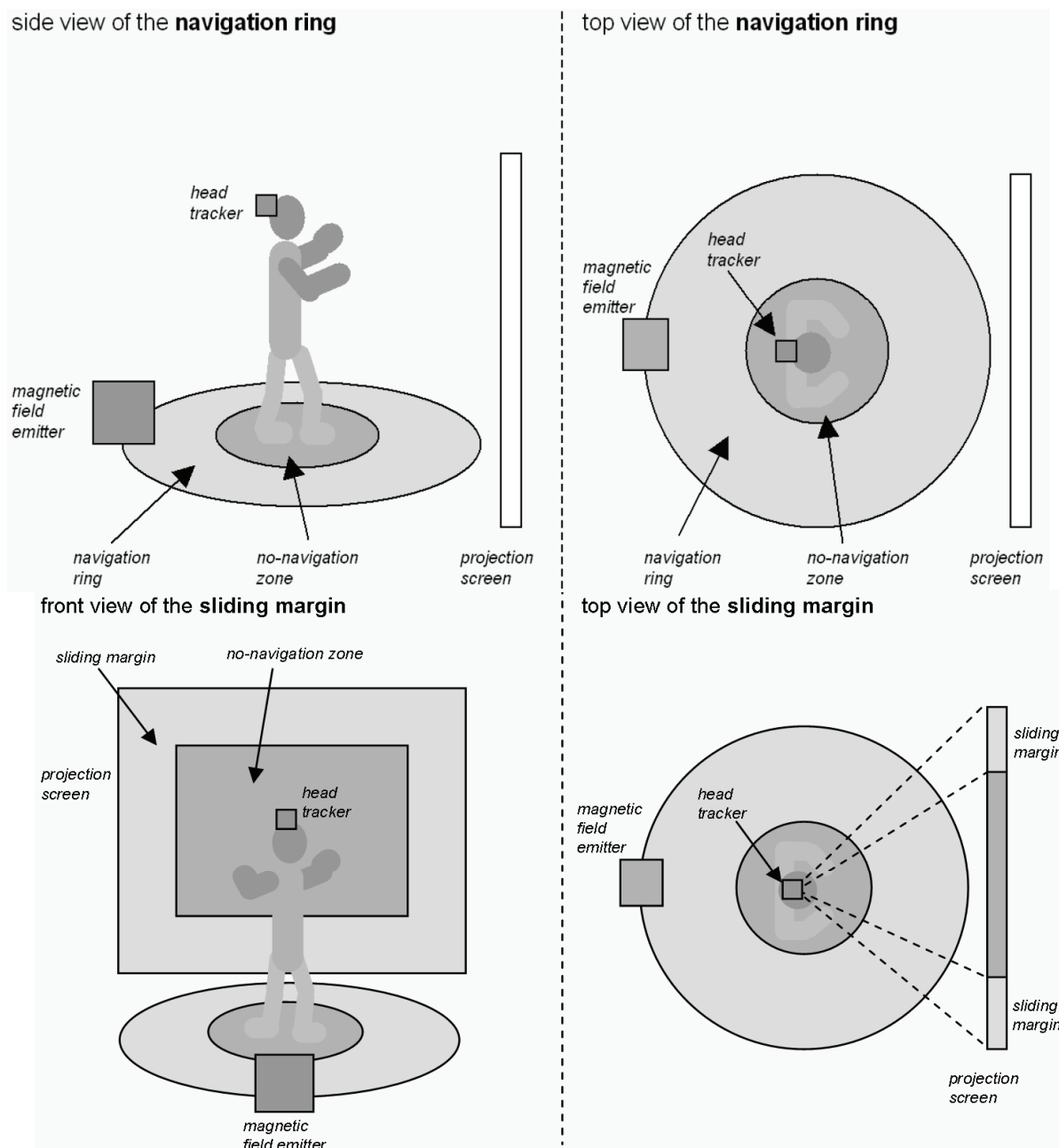


Figure 10.4: JUST VR navigation paradigm: “navigation ring” metaphor allowing for “walking around” the virtual environment; “sliding margin” metaphor allowing for “looking around” the virtual environment.

10.1.2 Scenario Authoring

As it has been already mentioned, handling an interactive scenario (as opposed to the interactive scene) is still an open issue. Nevertheless, for the JUST VR System, we elaborated our own semantics allowing for representation and semantics of the interactive scenario as a tree-like graph of states. During the execution, the scenario advances through the subsequent scenario steps. At each scenario step there is a set of actions available (defined by medical procedures) and the trainee must command his VA to execute one of them e.g. “Do chest compressions”, “Check responsiveness”, “Make mouth-to-mouth breathing”, etc. (see Figure 10.5).

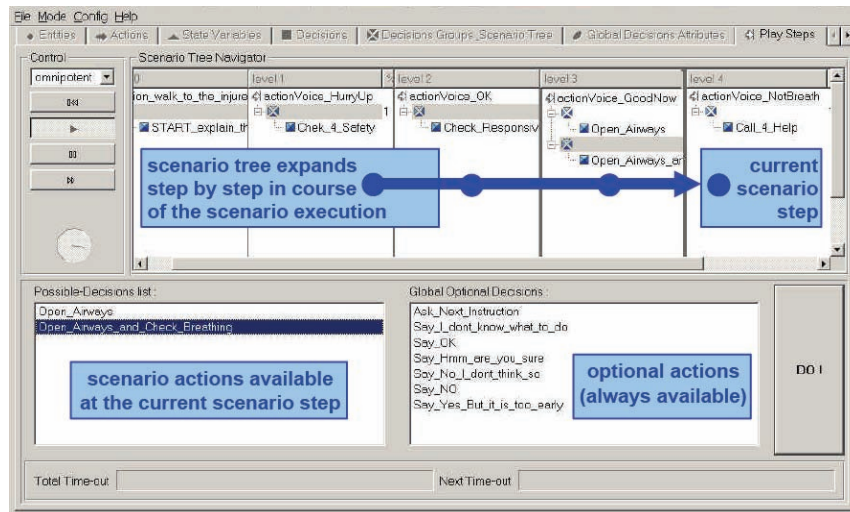


Figure 10.5: JUST VR system scenario execution: the main GUI used by the simulation supervisor to direct interactive scenario execution

10.2 Virtual Orchestra

The virtual orchestra application [Schertenleib2004] is an initiative to presents a semi immersive VR system based on a large projection screen designed to create an entertaining multimedia experience as shown in Figure 10.6. The end-user plays the role of the conductor of a virtual orchestra that performs in a 3D concert hall. One important aspect of this project is to compute 3D sound in real-time featuring dynamic acoustic. Moreover, the input device is not using the standard scheme of keyboard and mouse but rather relying on a PDA. The user control the simulation using the touch screen in a similar way than video games dedicated for the Nintendo DS handheld system [Nintendo-DS2004]. However, for increasing the sensation of presence and for providing more intuitive controls, a magnetic tracker was connected to the handheld device. With this setup, the user can freely move the handheld in the 3D space. This input scheme promotes similar interactions than the approach endorsed by Nintendo for their next home console [Nintendo-Revolution2006]. Here, we demonstrate that efficient integration of low-levels and hardware dependent modules like 3D sound generation and VR input devices managements can benefit from both a higher-level perspectives and concurrent programming model. The objectives of this work are to give to the music amateurs the opportunity to conduct a group of musicians and produce a new multimedia experience. The simulation is featuring several characters playing some instruments and reacting to the user impulse. The attention is ported on reproducing accurate 3D sound as well as providing simple interface for conducting an orchestra. This was made possible by partially capturing the expressivity of user's gestures without forcing them to perform a specific conducting pattern. During the design and development process, several challenges were faced to achieve a reasonable immersion for the users. Three main components were developed within the platform. The first two elements were low-level components handling the input device (magnetic sensors) and the 3D sound propagation. The 3D sound engine use is an extended work from the author's diploma master project [Schertenleib2002]. The challenge is to let the system kernel controlling the workflow of the sound processing using concurrent threads. Finally, the last component controls simulation data using python scripts. Being able to dispatch the tasks among priorities allow to give more processing power to specific elements. In this scenario, most of the attention is dedicated to provide a realistic acoustic experience. Thus, the system balancing is configured for this purpose. In effect, some studies made by Nvidia and the Microsoft developers teams during the conception of the Xbox's audio hardware [O'Donnel2001], have shown that the sound environment is so natural for humans that simulating it in a VR system gives impressive results, as the agent's –virtual characters- behaviors become more believable for the user. During the last couple of years, the synthesis and processing of 3D sound in VR applications did not receive enough consideration. This fact can be explained by the following reasons:

- Lack of space to physically locate the required speakers' configurations.
- The idea that the sound component of a VR system did not have the highest priority.
- Technical restrictions: The hardware required to compute real-time high-quality content on multiple speakers was expensive or inexistent.

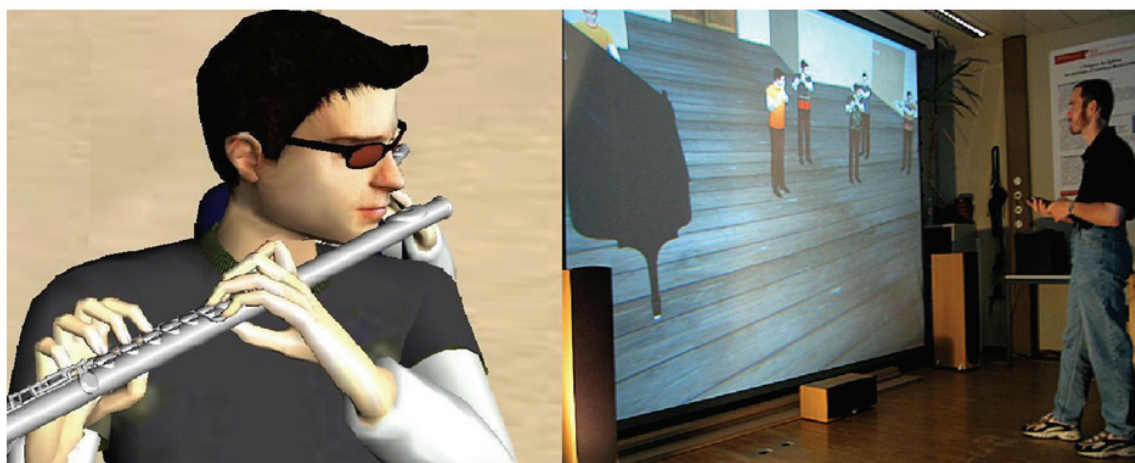


Figure 10.6: On the left, a virtual flutist following the user's tempo. On the right, an end-user interacts with the system using a PDA acting as an input device in front of a big projector screen.

10.2.1 Managing 3D Sound Processing

To handle the sound propagation from the initial position and orientation of sound sources and the listener, the system need to be able to analyze the 3D environment materials properties. In the virtual orchestra simulation, 3D sound environment is implemented through a customized version of the A* algorithm [IA-DEPOT, Rabin2002]. It gives us different points and angles of the objects, and then we can calculate all attenuations and effects to be set for filtering the wave. However, due to time-constraints, we cannot perform intensive computations for all sources, thus our simplified version allows some errors but guaranties a smooth performance [Schertenleib2004]. In other words, we do not compute every sound source at each animation frame, but check only if any drastic change occurs between two continuous steps. Otherwise, we compute extrapolations using the previous states of the sound source properties. Such operations are excellent candidates to be executed on a multiprocessing architecture. To take advantages of particular systems like PCs with dual processors or Hyper Threading processors [Binstock2003, Gerber2003], the sound component is relying on fine grain multi-tasking operations using the OpenMP API [OpenMP]. This increases the number of sound sources we can process in a single frame, based on a ratio of the number of logical CPUs available. Based on the system events, we analyze the data for controlling the human manager, which is responsible of synchronizing the sound processing with the human animation as depict in Figure 10.7. A extension of our approach was proposed by [Charlebois2005], which define that both the scene generation and sound handling can be integrated in the same 3D space. His concept intends to reproduce real objects with both graphical appearance and their sonic behavior.

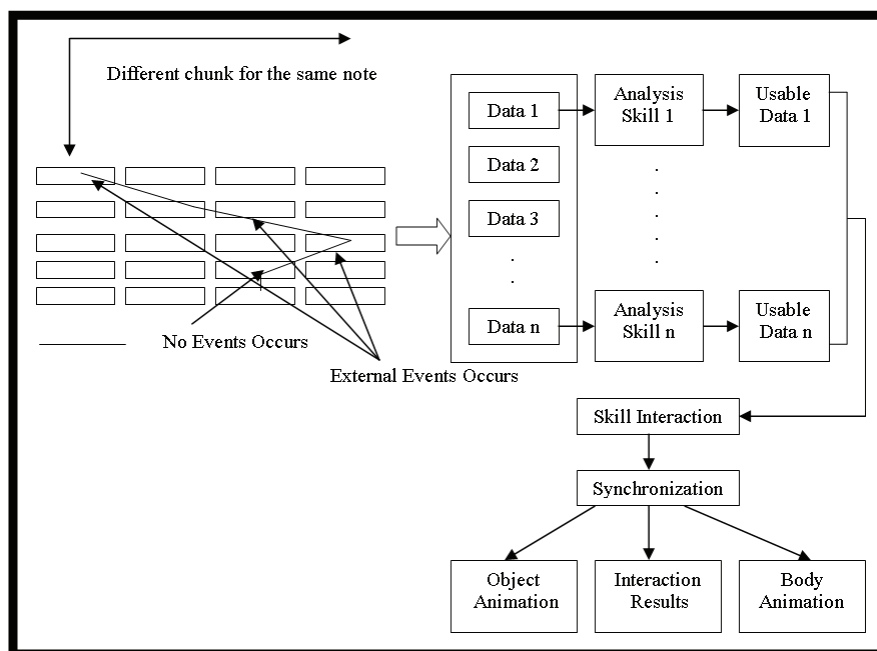


Figure 10.7: Example of a sound path depending of external events that control the synchronization between the sound rendering and the human animation.

10.2.2 Synchronization of Animations with Sound Propagation

3DRTS systems must react fast enough to events, to keep a good immersion. This is particularly true with the sound rendering, which is deeply affected by any delay in the processing. For the virtual orchestra, we need to synchronize the musicians' animations with the matching sound. Due to real-time restrictions, it is impossible to predict the transition of states, and thus we cannot prepare the right key-frame at a certain time. A careful management of the sound and animation databases is crucial to reduce latency [Arafa2002]. The processing is based on the work of Joaquim Esmerado [Esmerado2001]. We extended the work by adding more flexibility and independence on the musician attributes while relying on current hardware. Our system relies on preordering data using statistical information based on Markov chains. Some similar works have already been done in persistent worlds [Wilson2003] with the difference that they need to quickly transfer small amounts of information through network from a complex database. In our case, the data is locally stored but we must provide close to instantaneous access to the current state depending on external events. This involved managing a fast access multiple-level database. In our implementation, the high-level consists of direct interaction with the real database image by making requests to the hard-drive where the data is stored. However, as soon as a music partition is selected we can upload the full table in the main system memory. If we refer to the work of O'Donnel [O'Donnel2001]; a latency of 50ms or less is adequate for human beings while 100ms is the absolute limit for the sound propagation. In consequence, some constraints were enforced to deny varying the partition directly. Since we are working with a fixed music score, we approached the problem from the opposite side [Sonnenschein2001]. Instead of trying to follow the animation with the sound processing, we used the sound information to reflect and blend the right animation. A selected note within a music score will trigger the events allowing manipulating the different skeleton's bones of a specified musician [HANIM]. Therefore, as the system is managed through sound events, we can avoid delays in the sound processing; even though we will degrade a bit the animation smoothness. Human senses are more affected by sound distortion than by visual artifacts [G.N.A.G.].

10.2.3 Input Devices: a Virtual Orchestra Controlled by a Real Human

As stated before, the orchestra can be conducted through an interface implemented on a handheld device using a wireless link (IEEE 802.11b). There are two input channels: the 2D GUI on the PDA screen, and a magnetic tracker attached to it (see Figure 10.8). From the 2D GUI the user has a simplified view of the scene where he/she can edit the position and orientation of each musician –blue marks– in the scene. If the user

modifies the conductor's –green mark- position/orientation, the point of view of the scene in the projection screen is modified to match the new position/orientation of the virtual conductor. The corresponding modifications in the sound environment are computed and applied. In addition, this window allows playing/pausing the execution and selecting the scorebook to be interpreted by the orchestra. The Orchestra Direction window is one of two alternatives available for modifying the tempo - largo, adagio, default, allegro, presto- and dynamics –pianissimo, piano, default, forte, fortissimo- of the performance –the second method uses gestures tracking, details below. The user can just use the sliders, or, for the tempo, tap rhythmically with the stylus on the “conducting window”. The tapping frequency is mapped to one of the available tempo options. If the user stops tapping, the last tempo detected is applied.

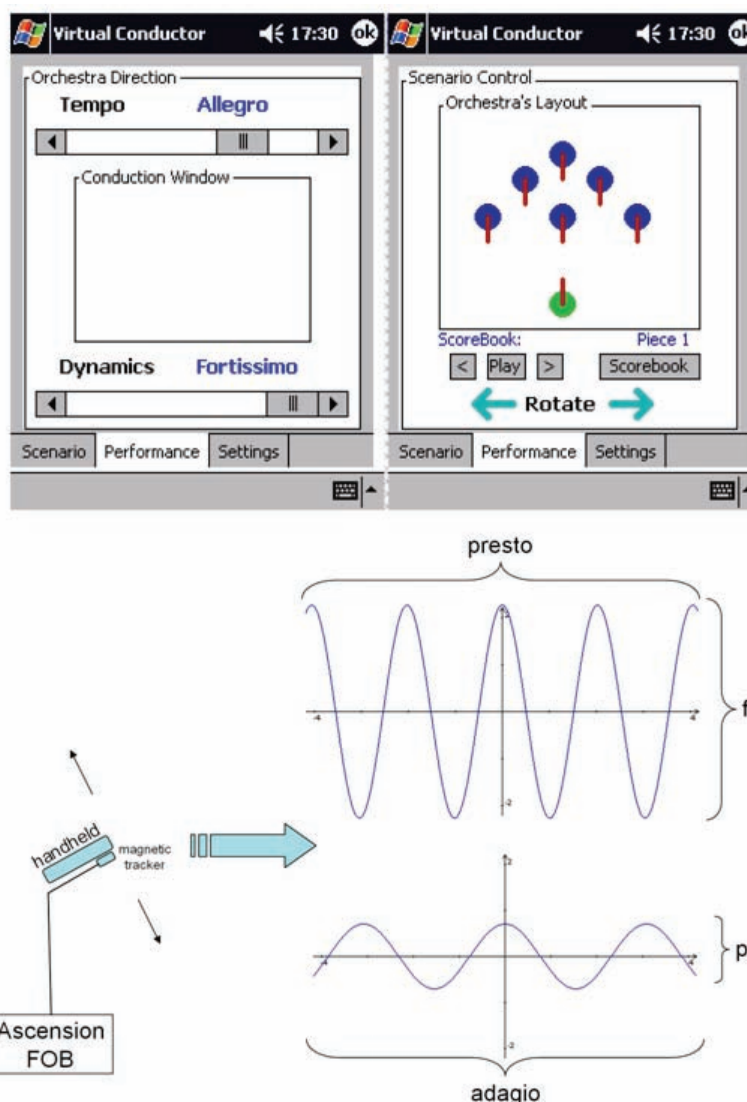


Figure 10.8: The handheld interface and interprets the data from the magnetic tracker.

The GUI-based tempo and dynamics control does not allow for interpreting the user emotions expressed through the arm gestures, the typical conducting gestures. To overcome this problem, we have attached a magnetic tracker to the PDA. The magnetic sensor is used to acquire the amplitude and frequency of the gestures performed while holding the handheld –a function of the rotation angles measured by the tracker to represent its orientation. The frequency is mapped into one of the possible values for the tempo, while the average amplitude of the gesture affects the dynamics –see scheme on Figure 10.8. The amplitude and frequency of the orientation values are calculated in real-time. The orientation values measured by the magnetic tracker are divided into 5 different regions corresponding to the possible values for the dynamics; the frequency is calculated as a function of the gesture's acceleration. The tracker (Ascension Flock of Birds) is sampled at 60Hz over a TCP connection. The information is processed on a dedicated PC –gestures analyzer- that sends through the network (TCP sockets) the current tempo and dynamics values to apply.

10.3 LIFEPLUS (AR Simulation)

The project LIFEPLUS was an attempt to recreate the ancient life of Pompeii in an augmented world. The simulation also feature realistic simulations of animated virtual humans actors (clothes, body, skin, face), who augment real environments and re-enact staged storytelling dramas (see Figure 10.9). The innovation combine the capability of a platform architecture dedicated for 3DRTS by adding specific modules for AR systems. One of those modules was responsible to track camera images dynamically. The goal of this project was to be able to setup the system in different locations and to let designers re-create simulations and characters behaviors using data-driven mechanisms. Moreover, as the computation for tracking images is a very intensive process, the system was taking benefit of the load-balancing using concurrent process for ensuring a smooth VR update.



Figure 10.9: LIFEPLUS: augmented reality scenario.

10.3.1 AR Life System Design

The key innovation of developing a dedicated module for AR simulations is confined around plug-in extra modules, which combine diverse technologies such a real-time rendering in AR. This include the additional support of real-virtual occlusion), real-time camera tracking, and dynamic behavioral scripting of actions. The integrated to the AR framework-tracking component rely on a two-stage approach [Papagiannakis2004, 2005]. First, the system uses a recorded sequence of the operating environment to train the recognition module. The recognition module contains a database with invariant feature descriptors for the entire scene. The runtime module then recognizes features in scenes by comparing them to entries in its scene database. By combining many of these recognized features; it calculates the camera location and thus the user position and orientation in the operating environment (see Figure 10.10). The main design principle was to

maximize the flexibility while keeping excellent real-time performance. The different components may be grouped into the two following main categories:

- System kernel components responsible for the interactive real-time simulation initialization and execution.
- Interaction components driving external VR devices and providing various GUIs allowing for interactive scenario authoring, triggering, and control.

Finally the content is classified into the two following main categories: a) Static and b) Dynamic content building blocks such as models of the 3D scenes, virtual humans, objects, animations, behaviors, speech, sounds, python scripts, etc.



Figure 10.10: The mobile AR-life simulator system (left). On the right, the real-time virtual Pompeian characters in the real site of the Pompeian thermopolium. Note the use of geometry ‘occluders’ that allow part of the real scene to occlude part of the virtual human (right).

10.3.2 AR Life Tracking

The runtime modules of our AR architecture is composed of the main engine responsible to assign time to the different modules that include virtual scene management and the analyse of real camera images. The dedicated AR service track 2D features between different frames and a signature module localize known features. The AR components classify these information into a scene database optimized for fast indexing. Then, a sequential solver will extract the 3D camera location. The last segment of this processing consist to blend both the real and virtual scenes using occluders and a two-pass rendering approach. These occluders are virtual objects matching real objects but that are not display in the resulting image. There are use in the first rendering pass to extract depth information of the real objects, so that we can occlude virtual object in the second rendering pass. The Figure 10.11 illustrates the AR component pipeline.

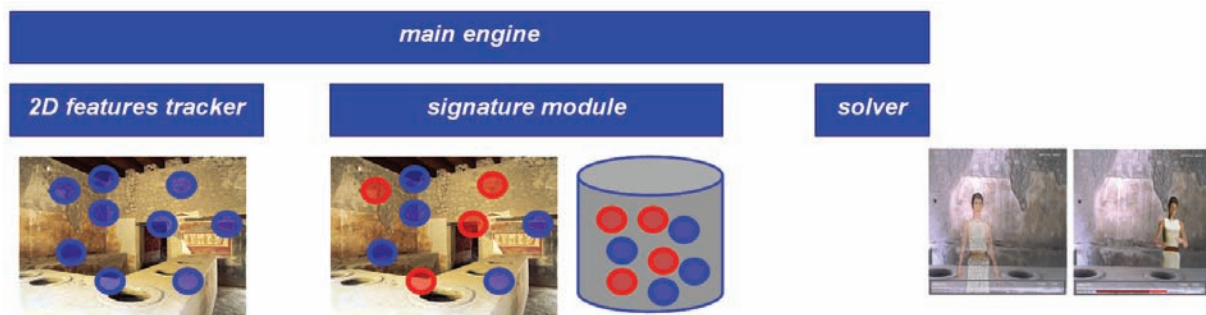


Figure 10.11: AR component pipeline.

10.3.3 MR Framework Operation for Character Simulation

The software architecture is composed of multiple software components called services; as their responsibilities are clearly define. They have to take care of rendering of 3D scenes and sound, processing inputs from the external VR devices, animation of the 3D models. In particular, complex animations of virtual humans including skeleton animations and respective skin and cloth deformations are managed. They are also responsible for maintenance and consistency of interactive scenario states controlled by python scripts at run-time. To keep good performance, the system utilized four threads. One thread manages the updates of all the services that we need to compute, such as human animations cloth simulation or voice (sound) management. A second thread handle the 3D renderer, who obtains information from the current scene graph about the objects to be drawn as well as the image received from the camera. It will change the model view matrix accordingly to the value provide by the tracker. The third thread has the responsibility of capturing and tracking images. The last thread is the python interpreter, which allows us to create scripts for manipulating our application at the system level, such as creating behaviors for the human actions (key-frame animation, voice, navigation). The rationale choice of a multi-threaded application model came first for keeping smooth human animations frame rate as well as using technological features like Hyper-Threading [MSDN-HT2004].

The AR system presented in Figure 10.9 features immersive real-time interactive simulations supplied with proper information in course of the simulation. That is why content components are much diversified and thus their developments is extremely laborious process involving long and complex data processing pipelines, multiple recording technologies, various design tools and custom-made software. 3D designers create virtual humans or auxiliary objects that are included in the virtual environments. Creating virtual humans require to record motion captured data for realistic skeletal animations as well as a database of real gestures for facial animations. Sound environments, including voice acting, need to be recorded in advance based on the storyboard. For each particular scenario, dedicated system data configurations specifying system operational parameters, parameters of the physical environment and parameters of the VR devices used are defined. In addition, scripts defining atomic behaviors of simulation elements, in particular virtual humans, drive the simulation events. These scripts can modify any data in use by the current simulation in real-time. This allows us to continue running the simulation whilst some modifications are performed.

10.3.4 Threading Model

Recent developments in CPU technology have extended the installed base of computers running multiple hardware threads simultaneously. For taking advantages of this advance in hardware technology, involves significant changes in systems design. Our approach is consistent and separate individual flow of information between threads, as describe in Figure 10.12. Every process has to be as independent as possible for reducing bottlenecks on data synchronizations. For instance, we clearly separate the 3D rendering and the AI or simulation events and scripts management. Today PC platforms that incorporate the Intel Pentium IV processor with Hyper-Threading or Dual-Core CPUs [Binstock2003] can run two hardware threads simultaneously. By exploiting three active threads, we are optimizing the resource usage. One might argue that using one more active thread than the number supported by the processors may be suboptimal, but it occurs the rendering thread is mainly bound on I/O operations, occurring when camera images are sent the frame buffer. By separating different components within our architecture, we are able to generate richer simulations, while keeping the same level of interactivity.

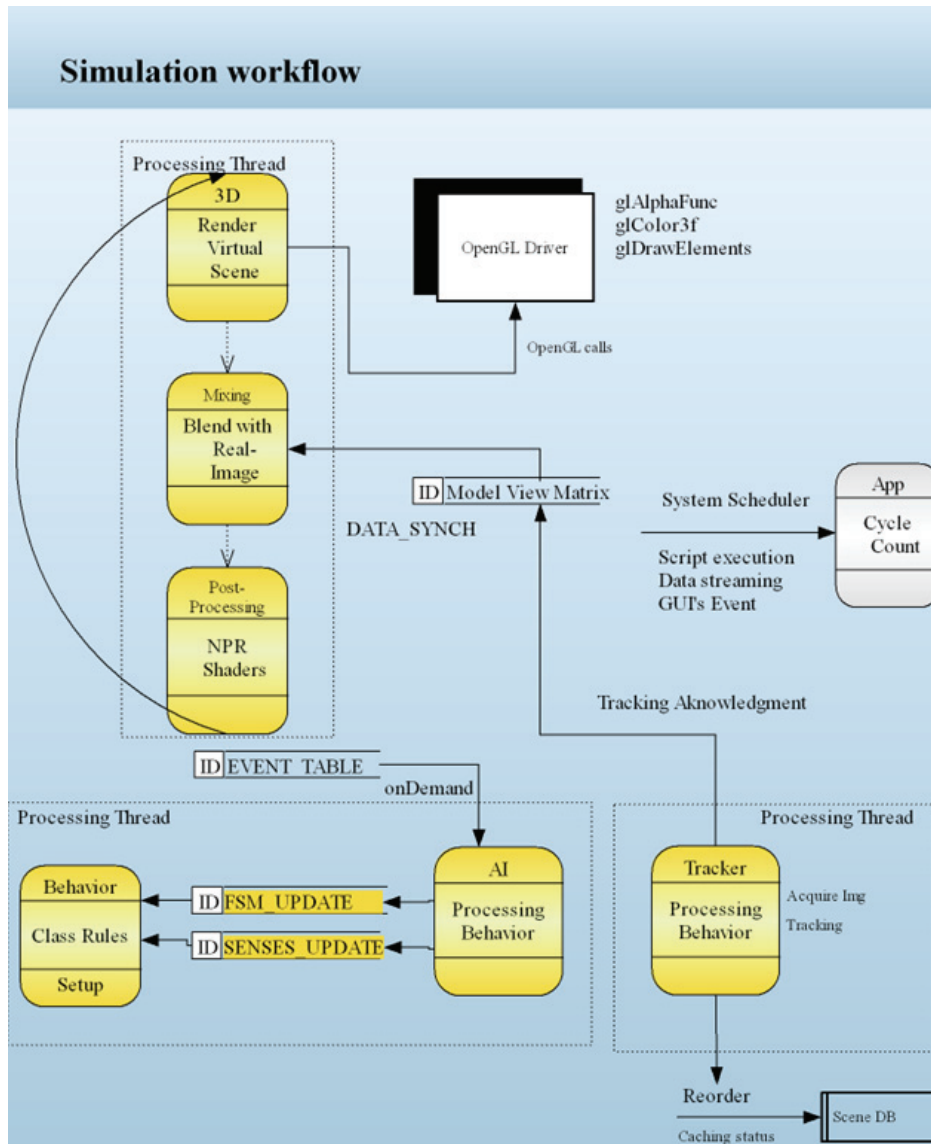


Figure 10.12: Threading model.

10.3.5 MR Registration and Staging

The LIFEPLUS AR system employs a markerless camera tracking solution for registering the CG camera according to the real one. This is a benefit since it eliminates the use of external tracking devices or avoids polluting the real scene with the use of known fiducial markers. However, the issue that arises is how to geometrically calibrate the camera and define the scene fiducial origin in world coordinates. Especially as our MR scenes have animated virtual characters, initial character staging, scaling and orientation is a crucial factor to determine correct initial, life-sized, believable geometrical registration [Papagiannakis2004]. However, for the MR real-time authoring stage, it is important that the complete staged experience can be initially positioned according to the real scene so the camera-tracking module can subsequently register it accordingly with the real scene. For the final geometrical registration, the Figure 10.13 illustrates the transformation sequence employed.

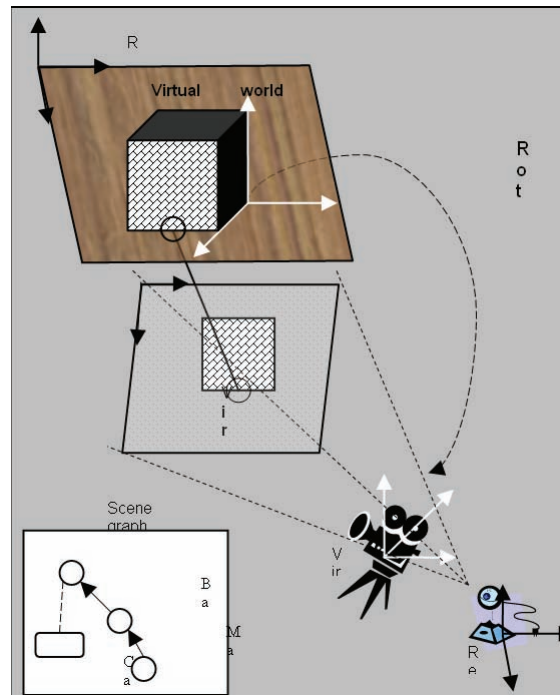


Figure 10.13: Projection and model view matrix transformation from the real camera to the VR scene.

The complete sequence of the processing thread as describe in Figure 10.12 is the sequence of the following steps:

1. Retrieve the camera image.
2. Run the feature tracker on this image.
3. Extract model view and Projection camera matrices.
4. Modify the combined camera matrix according to user authoring scaling/position controls (mapped virtual trackball mouse metaphor operation).
5. Apply the combined/adjusted camera matrix to scene graph renderer.
6. Move occluder geometries as root nodes in the scene graph with GL-DEPTH test set to OFF.
7. set the background image acquired from the camera in a 2D projected screen textured quad (thus since background image applied as a texture it is hardware accelerated and independent of window resolution, as hardware extension for non power of 2 textures was employed).
8. Modify the main scene root node according to user preference for further global positioning and scaling.
9. Render the scene.

Thus with the above simple and fast algorithm, we were able to both stage our fiducial MR characters and scene occluders without any performance drop in frame rate.

10.3.6 Interactive Objects Manipulation

Current standards for objects description [Collada, Web3D2004] rely on scene graphs containing some nodes to connect animations to external events. This is not enough to animate complex interactions, so the “Smart Object” paradigm was introduced by [Kallmann2001]. The primary idea is to define objects in term of their “interaction features” and “interaction plans”. This implied that simulation objects are seen as agents. To configure and control the direct and indirect interactions between agents, we rely on cooperative multitasking operations using microthreads [Abaci2006]. In the LIFEPLUS simulations, the virtual characters can interact with objects around them. Our primary concern was to address the practical issues that arise as results of incorporation of interaction-related information into a virtual environment. The object-interaction framework model attempt to solve these issues and is composed of the following components:

- A design tool that incorporates the definition of interaction-related information in the process of virtual object design.
- An XML specification for virtual objects, including appearance, animation and interaction aspects.
- An extended scene-graph that enables storage and query of interaction-related information at run-time.
- An event-based mechanism for controlling and coordinating animation of objects and virtual humans through scripting.

Let us consider an example where the virtual character needs to pick up a flask and carry it to somewhere else. In the simplest case, we will need to define two interaction attributes: The location the virtual character will stand towards the object when picking it up, and the hand placement and posture for grasping the object. The definition of interaction-related information can be complicated for a large virtual environment with many objects. Furthermore, some of this information inevitably depends on the geometry of the object. Therefore we have chosen to implement our design tool as a plug-in for 3D Studio MAX [Discreet]. Using our plug-in, interaction attributes can be created as the geometry of the environment is designed. This provides a consistent workspace for designers. In addition, it provides more efficient virtual environment management. Users of the design tool can select from various attributes and practical interfaces are provided for creation of each of these. For example, the hand attribute mentioned earlier can easily be created and each of the finger joints can be manipulated. An example is shown in Figure 10.14.



Figure 10.14: Virtual humans grasping virtual objects.

In this simulation, the virtual characters can execute basic actions. Multiple actions can happen simultaneously. Python microthreads control and coordinate animated elements in a flexible and concurrent manner. A script is responsible of one agent and can query and modify values of attribute and start agent actions. Scripts can also communicate to each other by events. Coming back to our example, a script controls the virtual character carrying the flask. This script continuously monitors the position and orientation of the hand of the virtual character and updates the position and orientation of the object as the character moves. For calculating the object's position and orientation, the object's predefined grasping postures for the hand is used.

10.3.7 Hardware Setup

Our case studies employed 5 fully simulated virtual humans [Papagiannakis2004], 20 smart interactive objects and one occluder geometry. The hardware configuration was relying on a single Alienware P4 3.20 GHz Area-51 laptop with a GeforceFX5600Go NVIDIA graphics card, an IEEE1394 Unibrain Camera [Unibrain2004] for fast image acquisitions in a video-see-through i-glasses SVGAPro [i-Glasses] monoscopic HMD setup, for advanced immersive simulations. The above hardware configuration (see Figure 10.15) boils down to 20fps for the camera tracker and 17fps for the main MR simulation.

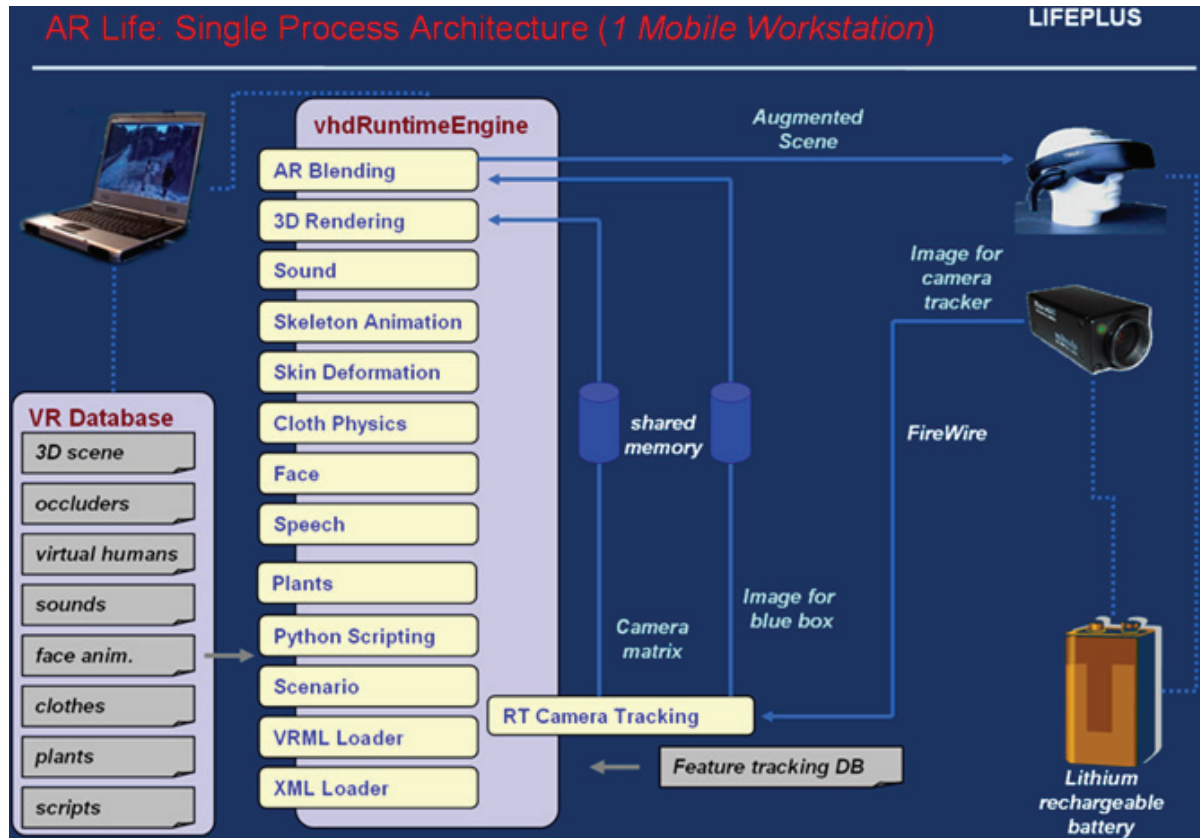


Figure 10.15: AR life simulator system.

10.4 Ogre3D Integration

Our architecture is not tightly connected to any specific component. In contrary to much 3D graphic centric architectures, our approach offer greater flexibility when replacing one component implementation. In this use-case scenario, the goals was to validate our method by substituting the OpenSceneGraph renderer by Ogre3D [Streeting]. Ogre3D is an efficient object-oriented graphic engine, which feature advanced 3D optimizations techniques such as hardware skinning using vertex shaders. In our test-bed simulation, we combine the Ogre3D renderer with our H-ANIM animation, AI, and scripting components. As Ogre3D offer the capability to create plug-ins for scene management, we were also able to integrate our ABT Tree implementation for early removal of objects. The Figure 10.16 illustrates a simulation featuring 700 H-ANIM compliant virtual humans equipped with the walk engine [Boulic2004]. In effect, the system versatility offers many interesting perspectives. For instance, we managed to run up to three different 3D renderer (Cosmo3D [Cosmo3D1998], OpenSceneGraph [Osfield], Ogre3D [Streeting]) simultaneously. All the simulation updates were compute once for all the three 3D renderer, sharing the same properties. This was made possible, as none of our component act as a master component driving the simulation workflow. This responsibility falls on the micro-kernel, which ignore the concrete implementation of modules.



Figure 10.16: Simulations using Ogre3D for the 3D renderer

10.5 Autonomous Virtual Human in Persistent Worlds

Here, we illustrate the research work of Etienne de Sevin [de Sevin2006], which promote real autonomy for non-player characters in order to live their own life in persistent virtual worlds. When designing autonomous virtual humans, the action selection problem needs to be considered, as it is responsible for decisions making at each moment in time. Actions selection architectures for autonomous virtual humans should be individual, motivational, reactive, and proactive to obtain a high degree of autonomy. The system implementation rely on the VHD++ framework, where an intensive usage of python microthreads offer the ability to describe a motivational model of actions selection for autonomous virtual humans which compute overlapping hierarchical classifier systems in parallel. By taking benefit of the cooperative multitasking offer by the python microthreads manager and the VHD++ framework gives a free distinct flow of control for each virtual character, where each of these characters can interact with the environment using objects or making path planning request in real-time. The Figure 10.17 illustrates the running application where both the 3D rendering window and the control and authoring tool gives feedback to the designers.

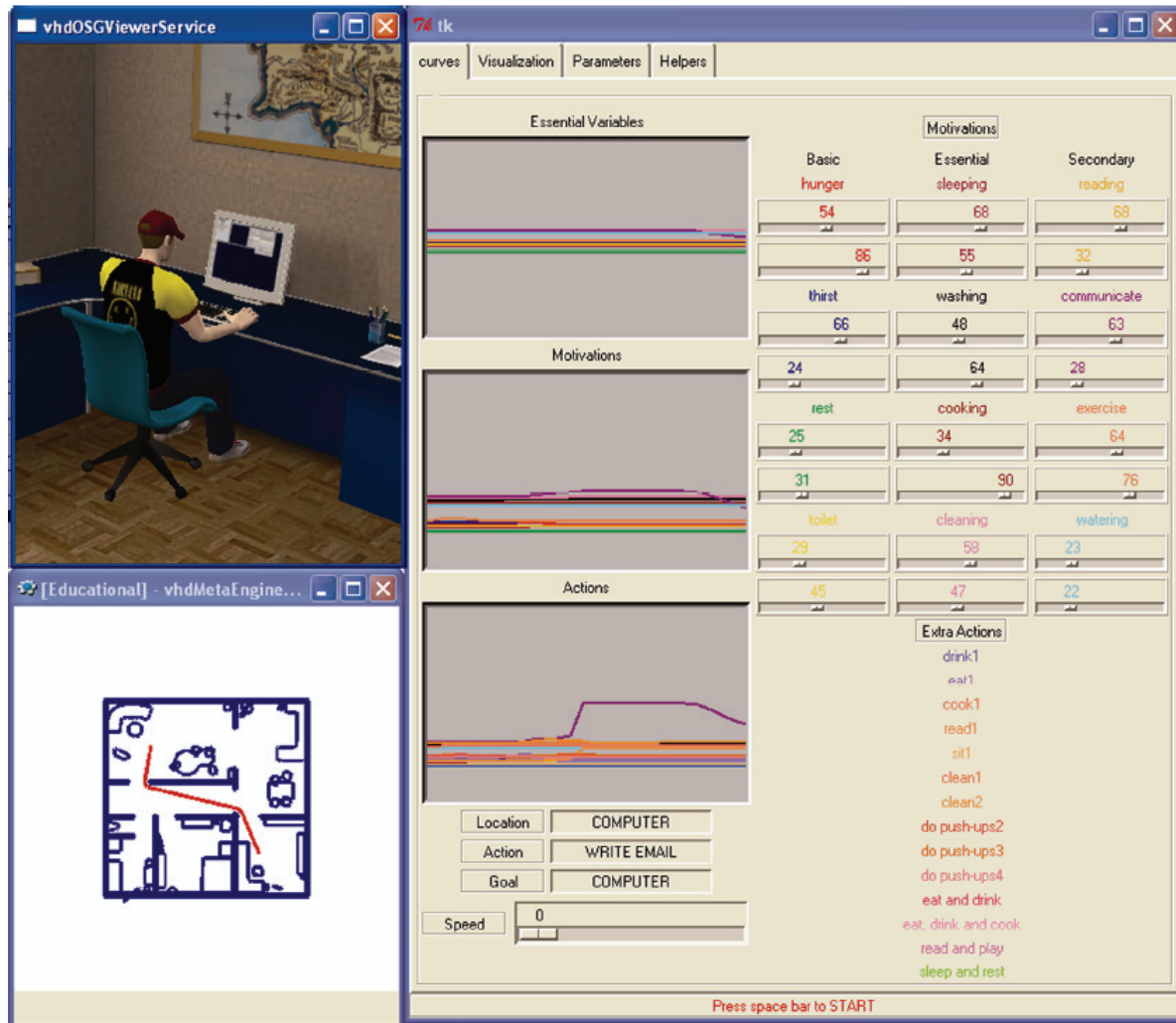


Figure 10.17: On the top left: the 3D window. On the bottom left: the navigation graph. On the right: controller for observing the evolution of motivation and set of triggers to change priorities dynamically.

10.5.1 Running Simulation into VHD++

Different test-bed scenarios were developed for analyzing the functionalities of the motivational model of actions selection. Here, virtual humans can live autonomously by perceiving their environments and satisfying different motivations at specific locations. The Table 10.1 describes twelve virtual human conflicting motivations altogether with their specific locations and associated motivation actions. Those constraints are dependant of the simulation context and therefore can be changed using data-driven mechanisms by uploading rules and constraints using python scripts. During the simulations, the VHD++ framework gives energy to each individual virtual character so he can choose the most appropriate action between the current set of conflicting ones according to his motivation and environmental information. Actions can be done at specific locations in the apartment. Compromise behaviors are possible, for example, the virtual human can drink and eat at the table. He can also perform different actions in the same place but not at the same time such as sleep or rest in the bedroom. Finally, the virtual human can perform the same action in different places: for example does push-up in the room or in the desk. Moreover, he has a perception system to allow opportunist behaviors. The distance and its importance in the decision-making can be defined at the beginning and during the simulation.

Table 10.1: All defined motivations with associated actions and their locations.

	motivation	location	actions
1	Hunger	Table (1)	Eat
2		Sink (2)	Eat1
3		Sink	Drink
4	Thirst	Table	Drink1
5	Toilet	Toiler (3)	Satisfy
6		Sofa (4)	Sit
7	Resting	Bedroom (5)	Rest
8	Sleeping	Bedroom	Sleep
9	Washing	Bathroom (6)	Wash
10		Oven (7)	Cook
11	Cooking	Sink	Cook1
12		Worktop (8)	Clean
13	Cleaning	Shell (9)	Clean1
14		Bathroom	Clean2
15		Bookshelf (10)	Read
16	Reading	Computer (11)	Read1
17	Communication	Computer	Communication
18		Room (12)	Do push up
19		Hall (13)	Do push up
20	Exercise	Desk (14)	Do push up
21		Kitchen (15)	Do push up
22	Watering	Plant (16)	Water
23		Table	Eat and drink
24		Sink	Eat, drink and cook
25		Computer	Read and communicate
26		Bedroom	Sleep and rest
27		Bathroom	Wash and clean
28	Default	Sofa	Watch TV
...

From a technical standpoint, the motivation model of action selection for an individual virtual human rely on exploiting the scripting mechanisms provided by the VHD++ framework and by providing GUIs to interface with the internal simulation objects. The simulation uses a path-planning module based on the work of [Kallmann2003], which enable obstacles avoidance when virtual humans navigate in specific locations using the walk engine from [Boulic2004]. The environment domain is defined using obstacles coordinates directly exported from 3D Studio Max [Discreet] into a XML file. Using the data-driven paradigm, users can not only define rules and goal-oriented behaviors but also alternate them dynamically during the simulation. The interaction with the internal simulation objects are automatically exposed by the framework on the scripting level, allowing to change almost all existing parameters and in particular each action effect factors increasing or decreasing internal variables. Then users can define a personality for the virtual human such as lazy, greedy, sporty, tidy, dirty... The scripted behaviors are directly accessible from python scripts using cooperative multitasking operations for dedicating one flow of control for each virtual character. This greatly simplifies the conceptual representation of agents for high-level designers. The real complexity comes with managing animations. Thus, animations are based on key-frames for each action including the interaction with objects. The key-frames need to be divided into three parts: one for preparing the action, one for the action itself and one for terminating it. In this case, the action key-frame can be interrupt with ease.

The results demonstrate that both the model and the framework are flexible and robust enough for modeling motivational autonomous virtual humans in real-time. The architecture generates as well reactive and goal-oriented behaviors dynamically. Its decisions making is effective and consistent and the most appropriate action is chosen at each moment in time with respect to many conflicting motivations and environment perceptions. Furthermore, apart from the low-level services provided by VHD++, all the behavioral rules and agents' management were developed into high-level interpreted languages and XML files providing the abilities to non-technical designers to develop safely 3DRTS in multitasking environments.

10.6 ERATO (Crowds)

ERATO is part of the European project [Erato], which intend to visualize and simulate virtual crowds in a reconstruction of a Roman Odeon in Aphrodisias in Turkey. The Odeon is populated with a virtual crowd experiencing a theater play. With the help of archeologists, architects, musicians and historians, we try together to give as an authentic experience as possible. The Odeon of Aphrodisias constructed in 2 AD, once the capital of Lydia in the Roman Empire, is located in the district of Karacasu south of Nazilli in southern Turkey. It is part of one of the most interesting archeological sites of the Aegean region. For more information on the Odeon reconstruction, see [de Heras-VAST2005]. The purpose of this project is to provide system architecture, where a crowd of virtual humans can interact individually, helping researchers, and historians to visualize and reconstruct heritage events. Some of the biggest challenges faced include providing tools for prototyping unique scenarios efficiently. Every such use case has it own constraints on data level or simulation events. We expect the system to be able to reconstruct any theater performance-taking place in ancient theaters in different locations where rites and habits are different. The Figure 10.18 showcase the Odeon and with the real-time interaction with the crowd.



Figure 10.18: Different viewpoints from the Odeon and the crowds.

Since the beginning of the project, the focus has always been on creating a graphical pipeline that could be applied to many different scenarios and cultural heritage settings. The goals were to overcome the development costs involved in creating unique individual characters. The solution is to promote variety in code and content reuse, by allowing to prototype quickly a cultural heritage application from scratch, using customizable meshes, textures, colorings, behaviors, and scenarios. Hardware constraints also oblige to keep memory footprint at reasonable levels, by using a limited number of mesh templates at a time. Each template serves for creating unique individual. By reusing the same geometry, we reduce the system memory usage as well as the number of 3D models to be designed [Gosselin2005]. However, using this method, the variety remains limited by the number of textures available. Our solution introduce colors modifications on the different body parts which composed a virtual character using an approach based on alpha maps as describe in [de Heras-VAST2005]. This high level of variety in crowds is necessary for creating believable and reliable crowds in opposition to uniform crowds. For a human crowd, variation can come from the following aspects: gender, age, morphology, head, kind of clothes, color of clothes and behaviors. However, the related freedom in the color palette selection goes against the realism of their historical counterpart. Within the EU project ERATO [Erato], we were able to organize different panels with archaeologists and historians, which gave us invaluable

information allowing restraining the color for the different body parts based on historical context as depict in Figure 10.19.



Figure 10.19: Representing and positioning social classes in the Odeon.

10.6.1 Resources Management

During the development, different iterations of the simulation were created. Some of the initial feedbacks received from our partners noticed that our humans were limited in the number of animations they could perform. In effect, it was a direct side effect of the way virtual humans rendering was handled. In our previous approach, full meshes were used as snapshots of the geometry, which could then have a single texture applied [de Heras-VSMM2005]. When we had around 10 templates and each one had 10 or so animations, we were already hitting the system memory limit of 2 GB per process under Windows or any similar 32-bit OS. Alternative solutions for creating and storing animations were designed which included hardware accelerated skinning algorithms, which reduce the memory consumption by around one thousand less memory [de Heras-VSMM2005]. Also for improving the system performance, the management of LOD was extended for modifying the frequencies for animations and behaviours updates based on LODs. This is achieving using an animations and agent's cache system. The cache represents a chunk of data that can be used by multiple instances of virtual characters at a given frame. This allows reducing the computation complexity by reducing the variety of both animations and behaviours on distant characters. In addition, we extend the animation variety by blending multiple animations using different parameters as depict in Figure 10.20.

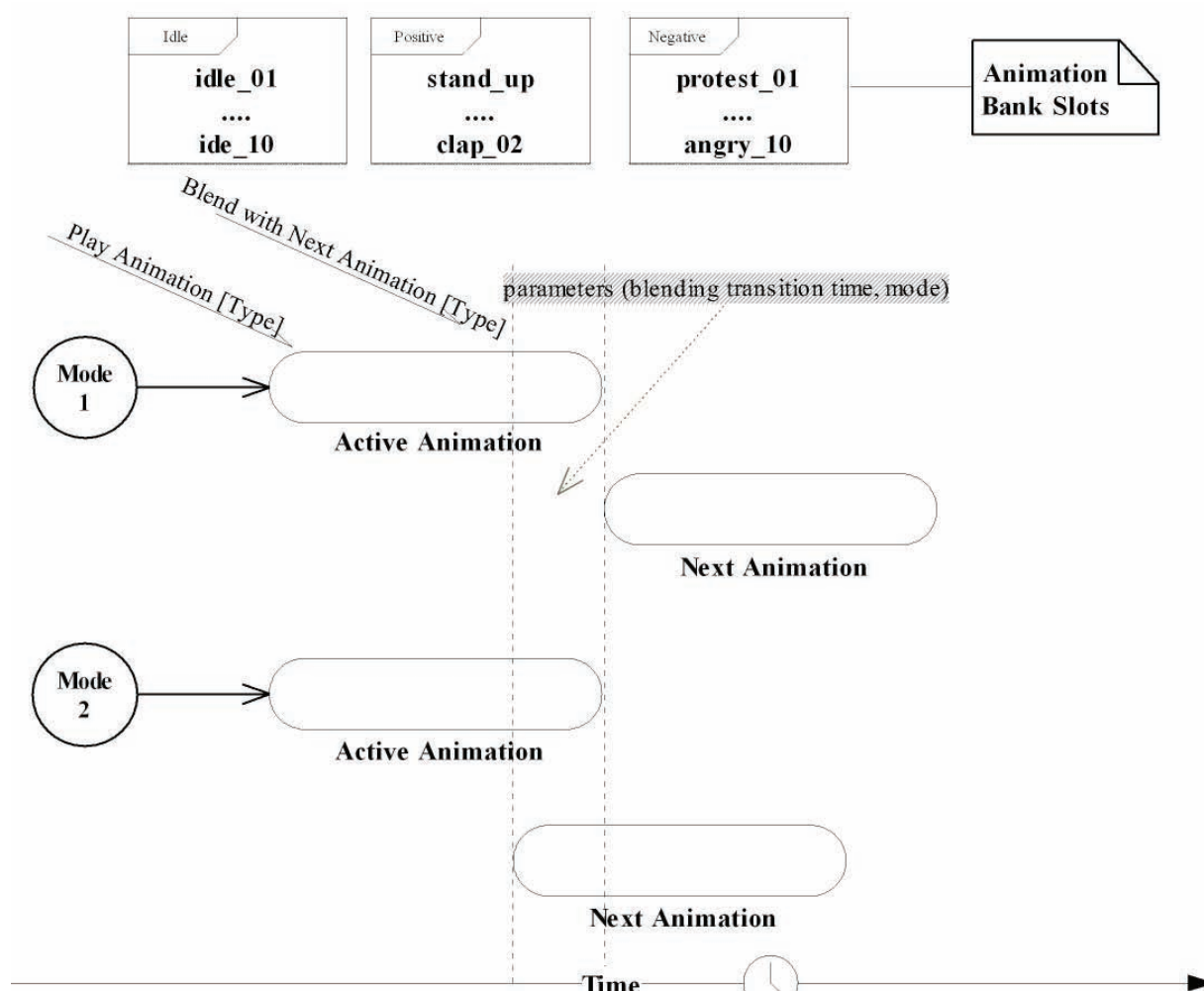


Figure 10.20: Animation blending pipeline.

10.6.2 Managing the Simulation Workflow

Crowd simulations have highly intensive resources requirements. As we expect to provide good interactivity levels with the simulation, we must assure a high refresh frequency for the graphic component. As our simulation goes beyond crowd rendering, we need to extend the design architecture by separating the 3D rendering and the AI processing into two different flows of controls as described in Figure 10.21. By keeping these two elements as distinct as possible, we are able to constrain the data synchronization issues coming with a multi-threading design scheme to a small subset of shared information. In [de Heras-VSMM2005], we describe how we exploit modern multitasking CPUs. Overall performance can be considerably improved by keeping every hardware thread active at all times. This is particularly true with I/O operations where the CPU is waiting for some acknowledgments, like blocking OpenGL commands for instance. Since we are decoupling the graphics components and the AI, we can generate behaviors that are more complex without sacrificing the overall level of interactivity.

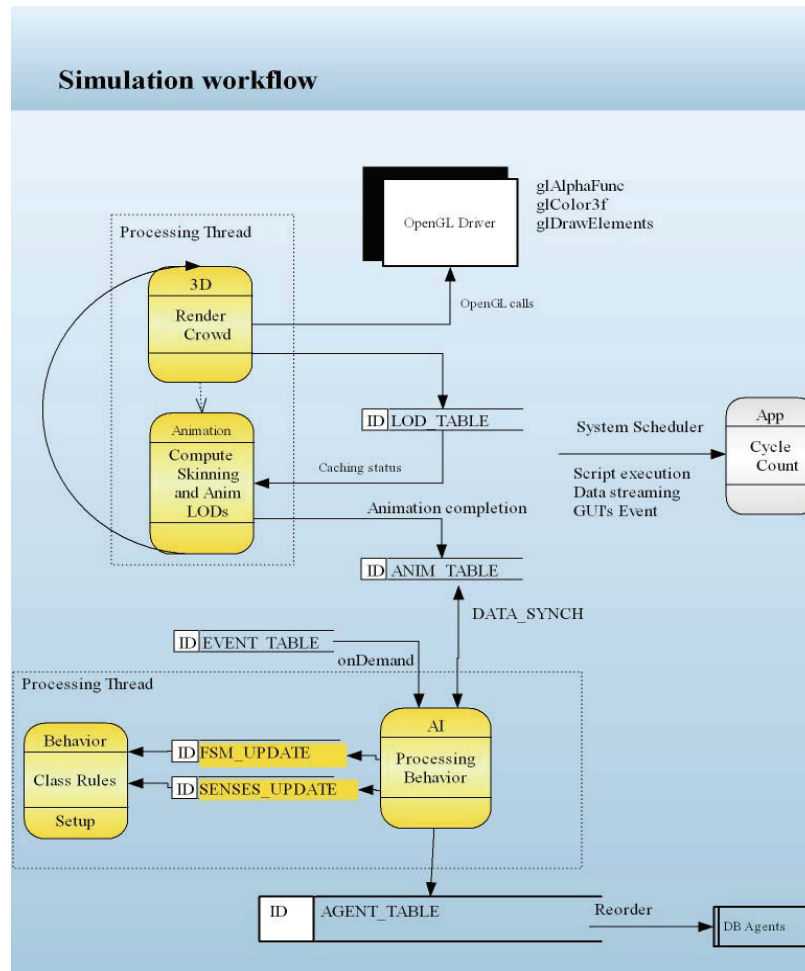


Figure 10.21: Simulation workflow that show the separation between the rendering and artificial intelligence execution threads.

10.6.3 Scenario Prototyping

To elaborate the different scenarios that can feature more than a few hundred distinct virtual humans, we need to provide tools and scripting capabilities to unleash the potential for simulation designers. Our system offers different levels of interaction for off-line and on-line simulation adjustments. Therefore, our applications are able to parse scenarios configuration files written in the scripting language Python. An important part of our scenario is the distribution of the audience in the Odeon. We created a plan of the distribution according to historical sources. This allows distributing the plan of the audience according to social classes of the people (see Figure 10.19), starting from the center there are places for nobles, patricians and plebeians. We take advantages of the particular characteristic of ancient theaters that differ from nowadays theaters. In effect, the audience were seating on stone stairs. This provides more freedom to setup the crowds especially that we expect to place the virtual humans in a non-uniform manners. To facilitate the positioning of digital actors, we created a grid of valid position in the theatre following the distribution of the seats that match the historical circular geometry architecture of our models. From a higher perspective, every layer of seats becomes a circular shape with different angles and radius. We have developed some routines to distribute the seats among those levels. To avoid that every character faces the center of the theater, we have introduce Perlin noise [Perlin1985] in their facing direction, providing a more believable and realistic setup.

10.6.4 Scenario Management

Most of our use-case scenarios were based on reconstructing life in ancient theaters. The Table 10.2 describes the sequence and user interactions of a complete scenario.

Table 10.3: Scenario workflow.

Part	Description	User control	Interface	Action	Elements
1.0 Introduction (sequence)	Audience idle -> senators entering, ...sit -> people stand up and salute -> when sequence is over, go to Part 2.0	> predefined camera <u>anim</u>	= no interface	Script (trigger at the same time): - Senators entering + crowd reacting - synchronized sound (clapping) - synchronized camera path	# <i>scenarioPart1.0.py</i> # <i>Part1.0_intro.wav</i> # <i>eratoVP_intro.path</i> = all synchronized (duration 50sec)
2.0 Selection (interactive loop)	User chooses what he wants to see Audience idle = loop	> <u>control the camera</u> - choose cameras > <u>choose simulation</u> > <u>quit</u>	= 4 buttons (1 per camera) = 3 - theatre play => go to 3.1 - music => go to 3.2 - spray => go to 4.0 = 1 button (x)	- Switch to selected camera & corresp. sound (current time) - Switch to selected scenario&sound (start time), for current camera - quit app.	# <i>scenarioPart2.0.py</i> # 4 loop sound: <i>Part2.0_idle*.wav</i> # 4 cameras: <i>eratoVP_crowd*.path</i>
3.1 Theatre Play (sequence)	User hears actors playing on stage -> Audience is reacting to the play: -> when sequence is over, go back to 2.0	<i>Idem 2.0</i>	<i>Idem 2.0</i>	<i>Idem 2.0</i>	# <i>scenarioPart3.1.py</i> # 4 sounds: <i>Part3.1_CreonCrowd*.wav</i> 4 sounds: <i>Part3.1_CreonStage*.wav</i> = synchronized with script (1min45sec) # 4 cameras: <i>eratoVP_crowd*.path</i>
3.2 Music (sequence)	User hears music played on stage -> audience listening, -> at the end applauds -> when sequence is over, go back to 2.0	<i>Idem 2.0</i>	<i>Idem 2.0</i>	<i>Idem 2.0</i>	# <i>scenarioPart3.2.py</i> # 4 sounds: <i>Part3.2_musicCrowd*.wav</i> 4 sounds: <i>Part3.2_musicStage*.wav</i> = synchronized with script (1min02sec) # 4 cameras: <i>eratoVP_crowd*.path</i>
4.0 Spraying Emotions (interactive)	User "sprays" emotions on the crowd - default crowd status = idle (<i>Idem 2.0</i>)	> <u>choose emotion-spray</u> (default = happiness) > <u>spraying emotions</u> > <u>go back</u> > <u>quit</u>	= 2 - spray1: happiness - spray2: sadness = mouse = 1 button (←) => go to 2.0 = 1 button (x)	- Switch to selected emotion for current camera - Spray... - Switch to 2.0 + current camera - quit app.	# 2 spray scripts (emotions).22. # 3 loop sound files emotions <i>Part4.0_laugh.wav</i> <i>Part4.0_cry.wav</i> <i>Part4.0_idle1.wav</i> = loops (to mix according to percentage sprayed...) # 1 camera: <i>eratoVP_crowd1.path</i>

10.6.5 Simulation Controls

To control the simulation events, our software architecture is responsible for maintenance, consistent simulation, and interactive scenario states controlled by python scripts at run-time. Our system relies on microthreads for spreading the workflow and scripts execution over several frames. This offers the ability to describe complete sequences of events as Python scripts. The following listing is an extract of a script sequence that affects behaviors of digital actors dynamically.

Listing excerpt from a scenario:

```
import time

#music has started: people listen
for i in range(len(agents)):
    agentName = agents[i]
    simulationService.setCurrentState(agentName , "State_ListenMStart")

while time.time()-val < 45:
    #wait for the completion of the music score.
    vhdYIELD

#concert finished: applause
for i in range(len(agents)):
    agentName = agents[i]
    simulationService.setCurrentState(agentName , "State_PositiveStart")

while time.time()-val < 57:
    vhdYIELD
#back to idle
for i in range(len(agents)):
    agentName = agents[i]
    simulationService.setCurrentState(agentName , "State_IdleStart")
```

In effect, virtual characters are associated to animation banks, which define sets of animations connected to human emotions such as being positive, negative, laughing, crying. These emotions are represented as distinct states within an HFSM system where states are defined using Lua metatables [Ierusalimsky2003]. The different scripts which handle the different sequences of our scenario and which trigger sounds based on the current viewpoint are controlled by the end-users through dedicated widgets, as illustrated in Figure 10.22.

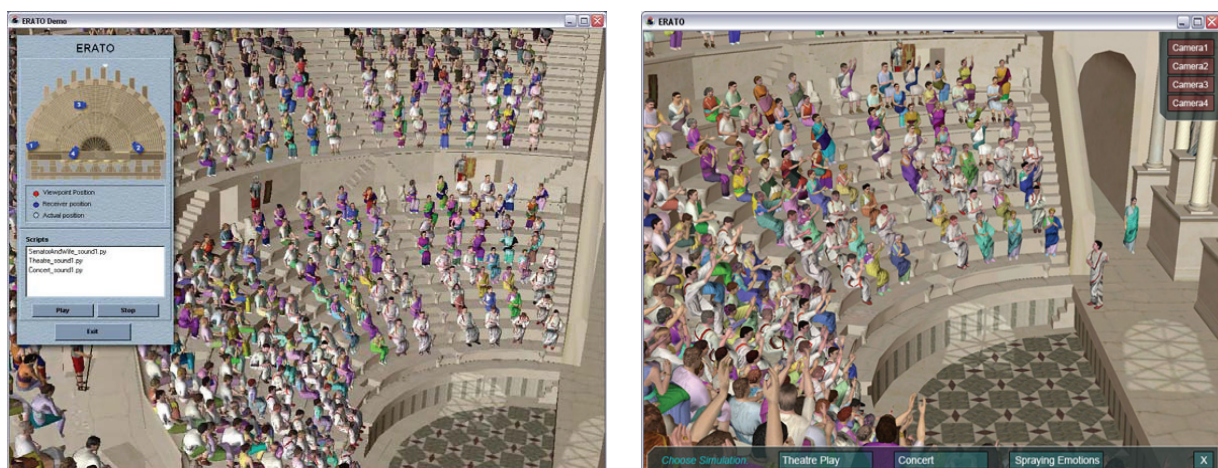


Figure 10.22: Interaction with the crowd is possible through using predefined scenarios and camera viewpoints.

To conclude, Figure 10.23 illustrates the different widgets available at run-time by our system. The application provides different set of widgets for both end-users and designers to interact with the crowds in real-time. To illustrate the simulation complexity, our most complete use-case scenario features around 800 virtual humans including two hi-fidelity actors representing two roman senators and 8 different humans' templates. The animation bank is composed of 750 unique animations. All of these assets are directly manipulated by simulation events relying on more than 50 python scripts similarly than in [Papagiannakis2005].



Figure 10.23: Global application overview.

10.7 vCrowds

The vCrowds application highlights the integration of crowd simulations within the framework. In this particular scenario, a few thousand virtual characters are managed using behaviors and navigation graphs [Pettre2005]. The system adds semantic information to different areas within the virtual world. To overcome the limitations of restricted resources for building crowds applications in an urban environment, the vCrowds application generate and use many different walking and running animations based on the work of Glardon [Glardon-CASA2004, Glardon2004]. By definition, virtual humans do not have the capacity to think by themselves. For this reason, much time and efforts is spent in finding solutions to simulate their knowledge and behaviors. In the specific context of real-time crowd simulations, the task becomes even more difficult: the spectator needs to believe in the uniqueness of every single virtual human, and thus varied behaviors are essential. In this application, the virtual humans come with knowledge of their environment by exploiting a navigation graph: for a given environment is created a set of nodes, representing navigable areas, connected with edges, corresponding to the paths between these areas. Those nodes are label with simple semantic information relative to the region covered by the node, such as Park, Hotel, Circus, etc. (see Figure 10.24).

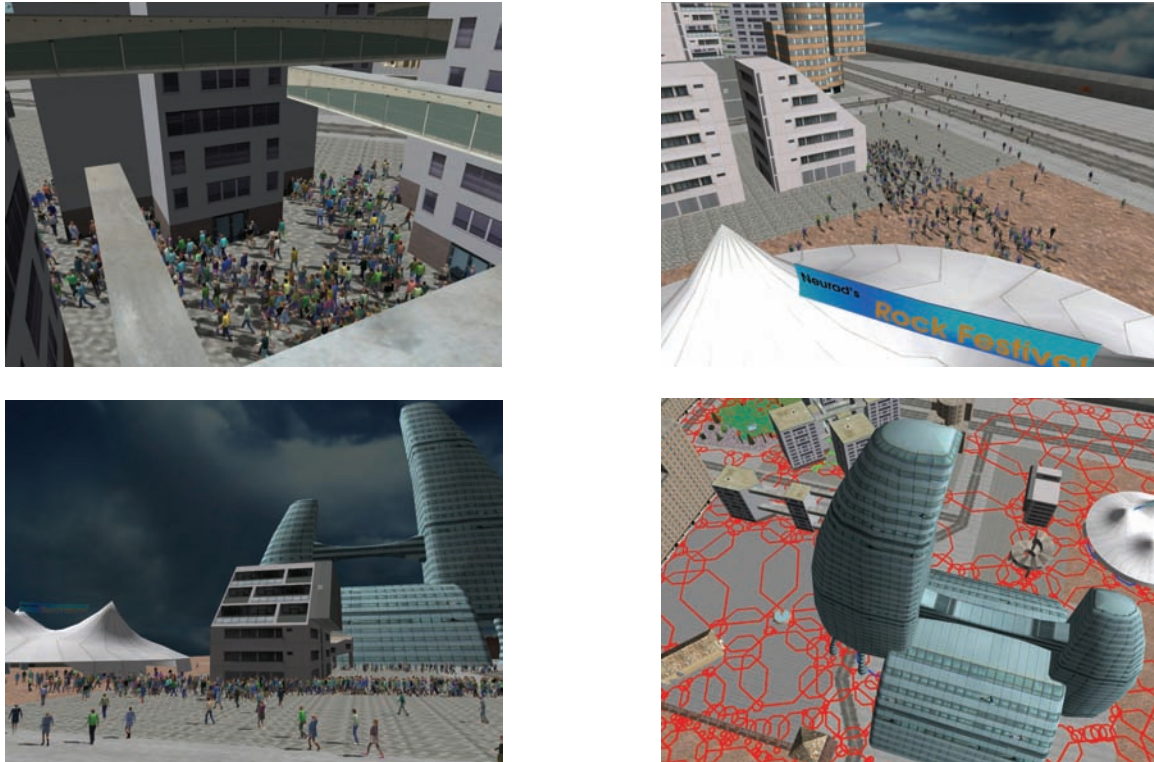


Figure 10.24: Crowd simulation within an urban environment.

10.7.1 Navigation Graph

The evolution of a crowd within an urban environment implies many emerging navigation issues:

- The path planner receives many path requests that must be answered in real time. Indeed, we cannot afford to have virtual humans lingering and waiting for a path.
- The paths that the virtual humans follow need to be varied so that those going to the same place will not follow the same list of points.
- The virtual humans cannot afford to stop at exactly the same position once they have reached their destination.

These constraints are controlled using semantic information and the abilities of both the walk engine and the navigation graph capabilities to offer more flexible navigation paths [Yersin2005]. The data structure relies on representing navigable areas under the shape of small vertical cylinders. Areas outside all cylinders are forbidden and thus, going from a cylinder to another is possible only if they are intersected. Since the navigation graph detects where obstacles lie, this structure allows to delimit the borders of a building for instance (see Figure 10.25).

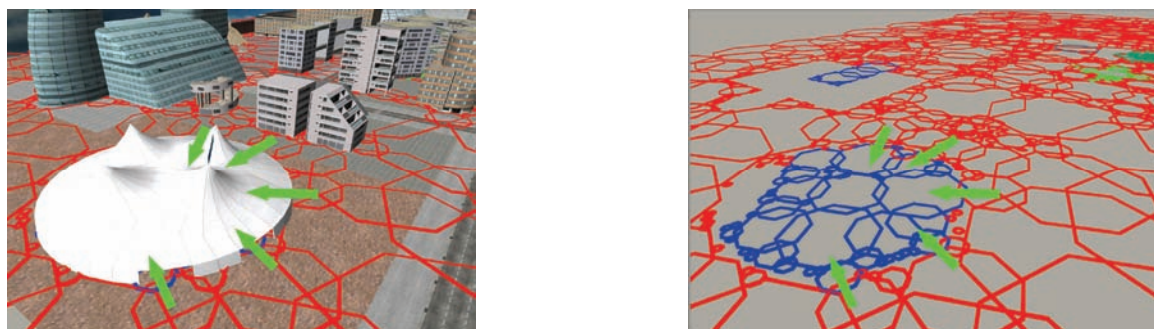


Figure 10.25: On the left, the circus with its 5 entries and on the right, its nodes (delimited with blue) connected with the rest of the graph only at these 5 entry points.

10.7.2 Animation Variety

The multi abstraction layer representation for the virtual humans data structure allow to provide high-level mechanisms where the designers can assign tasks to individual characters which will dispatch request to lower levels in the hierarchy including path finding and animation modules. For instance, the path planner with subdivide tasks into sub-goals such as set of ascending targets points. Each of these sub-goals is becoming a FSM where every state is defined as a Lua metatable (see Figure 10.26).

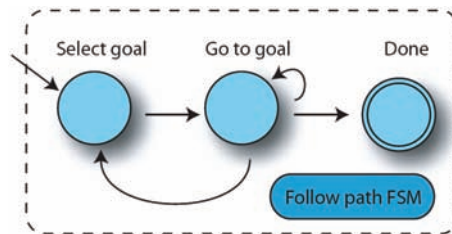


Figure 10.26: Navigation FSM: the path is composed of sub-goals based on the navigation graph.

On the animation side, Glardon et al. [Glardon2004] have introduced a PCA-based walk engine capable of animating on the fly human-like characters of any size and proportions by generating complete locomotion cycles. They have captured walk and run motions from several persons, from which they have created a normalized model. There are mainly 3 high-level parameters that allow modulating these motions:

- Personification weights: 5 people different in height and gait have been captured while walking and running, one of them being the author. This variable allows the user to choose how he wishes to parameterize these different styles.
- Speed: the 5 different people mentioned above have been captured at many different speeds. This parameter allows choosing at which velocity the walk/run cycle should be generated.
- Locomotion weights: this parameter defines whether the cycle is a walk animation, a run animation, or a blend between them.

Thus, the engine is able to generate a whole range of varying walk/run cycles for a given character. To efficiently animate the locomotion, i.e., walk and run motions, of each individual, we generate a certain number of locomotion cycles for each human template as preprocess. In this particular scenario, we have generated more than 1000 different locomotion cycles. Here, we do not use the global translation of the generated cycles, because we have found an advantage to proceed without it in the current framework, i.e., we have more freedom to steer the virtual humans. The main drawback to not use the global translation is that we have to take care of limiting cumbersome foot sliding effects. We sample for each template a set of more or less fast walk cycles and for each of them we sample a set of different locomotion types. Though we do not yet use the personification parameter while generating the walk cycles, we already perceive a sense of variety in the way the crowd is moving. Virtual humans walking together with different locomotion styles add to the realism of the simulation. Generating various walk cycles produces a huge amount of data to store and manage. To alleviate this complexity, we use an embedded database [Sqlite] to store all the animations used during the simulation. There are several advantages of using such a database: the data is efficiently packed and more importantly, it can be augmented with meta-data, e.g., the duration of the animation, its type and to which template the animation applies [Lange2003, Lee-GDM2003]. For instance, a locomotion animation can be efficiently stored along with its speed and style.

10.7.3 Multithreading for Crowd Simulations

To take advantages of modern hardware technology involves significant high-level modifications within the system design. [Kruszewski2005] have shown that providing an architecture that can scale into dedicated processes will allow creating crowd simulations that are currently only possible in a non interactive mode. Our approach is based on separating the flow of control between three different threads, as described in Figure 10.27. Each of them is responsible for one specific component. For instance, we use a dedicated process for the 3D rendering, another one for the AI processing, and a third one for the event handling and script executions. To maximize the benefit of our multi-threaded approach, we need to keep the shared data as minimal as possible.

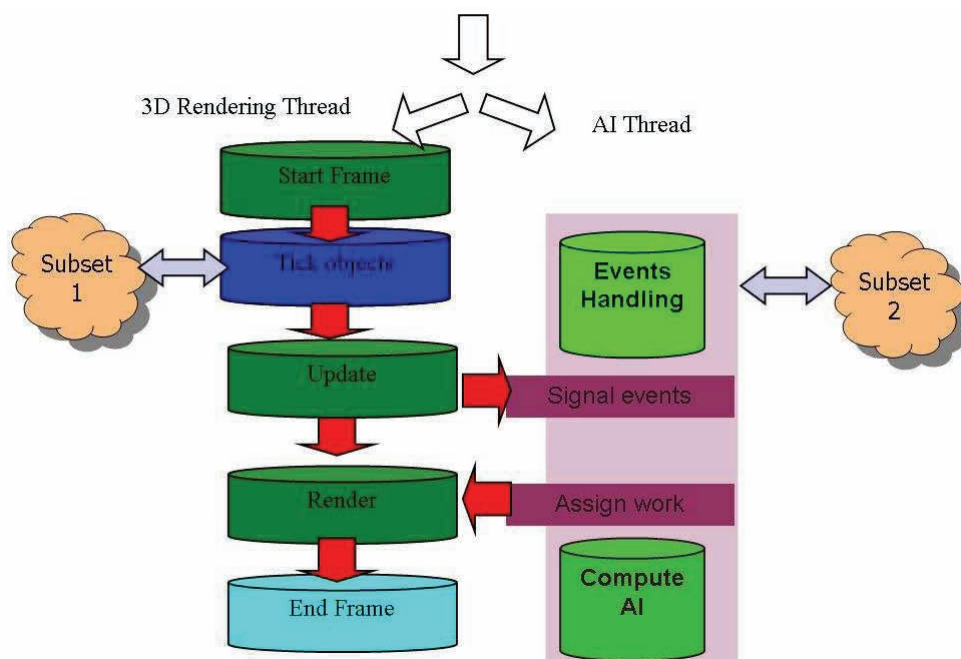


Figure 10.28: Threading workflow.

Since most current PC platforms can run two (or more) hardware threads simultaneously, by exploiting three active threads, we are optimizing the resource usage. One might argue that using one more active thread than the number supported by the processors may be suboptimal, but it occurs that the rendering thread is not running at full speed, being bound on synchronization points. The Figure 10.29 illustrates two issues affecting synchronization points between CPU and GPU. This system architecture and performance also scope better with current trend in CPU hardware design, which features multiple simpler in-order cores [Wloka2003, Brown2005, Hofstee2005].

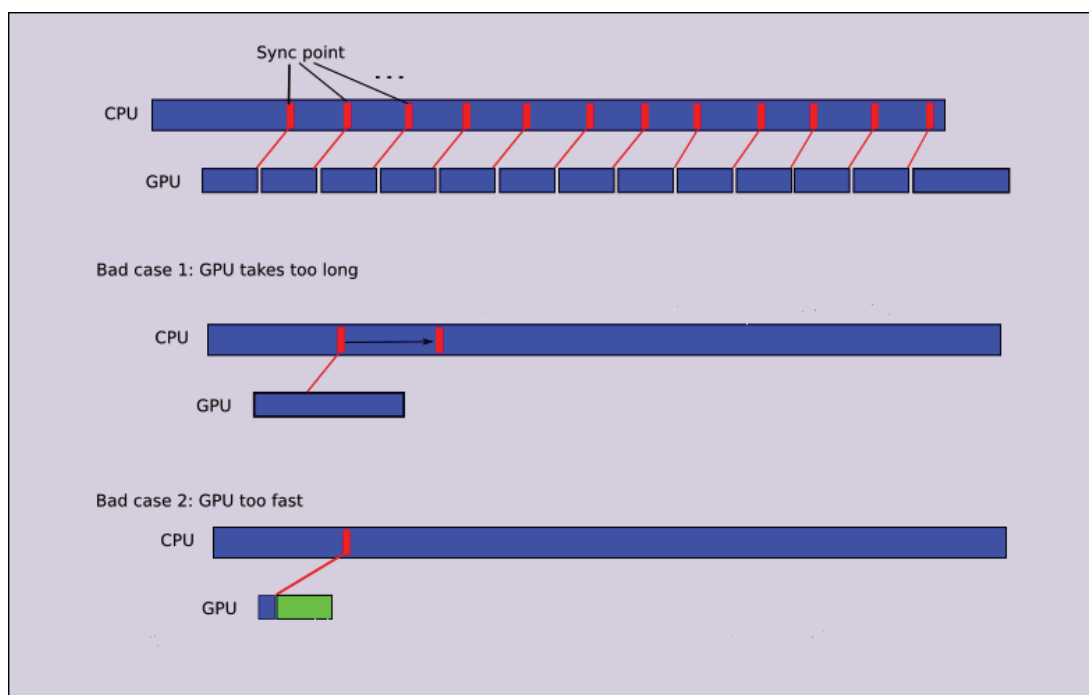


Figure 10.29: CPU/GPU synchronization points.

10.8 Other Use Cases

Finally, we conclude this section by describing the interoperability of our architecture with different assets and simulations. The images **10.30A** are screenshots from preliminary results obtained with the [EPOCH] project. EPOCH intends to bring together the combined expertise and resources of technologists, heritage administrators, heritage professionals and communication experts concerned with the effective and sustainable application of digital technology to archaeological research and cultural heritage presentation at museums, monuments, and historic sites. The contribution of VRLab consists to animating the virtual monks and abbot populating the abbey of ENAME around 1050 A.D. Images **10.30B** illustrate the 2D rendering capabilities of our framework. In this occurrence, we use 2D shapes to analyze steering behaviors in real-time. Images **10.30C** illustrate the interoperability of content using game assets (on the left a map from the game Quake 3, on the right a map from the game Half-Life). This gives the abilities to reuse existing content and to integrate them within our framework. In this example, the interoperability was done by creating a Quake 3 map loader plug-in for OpenSceneGraph. The images **10.30D** illustrate the interoperability of our urban crowd simulations with a 3D reconstruction of a 760-acre district from the city of Qjiang made by [3dvri]. The model complexity in its binary form (.ive) represents 290MB using all optimizations (DXTC-5 texture compression, optimizer...) and feature 1'437'243 polygons.

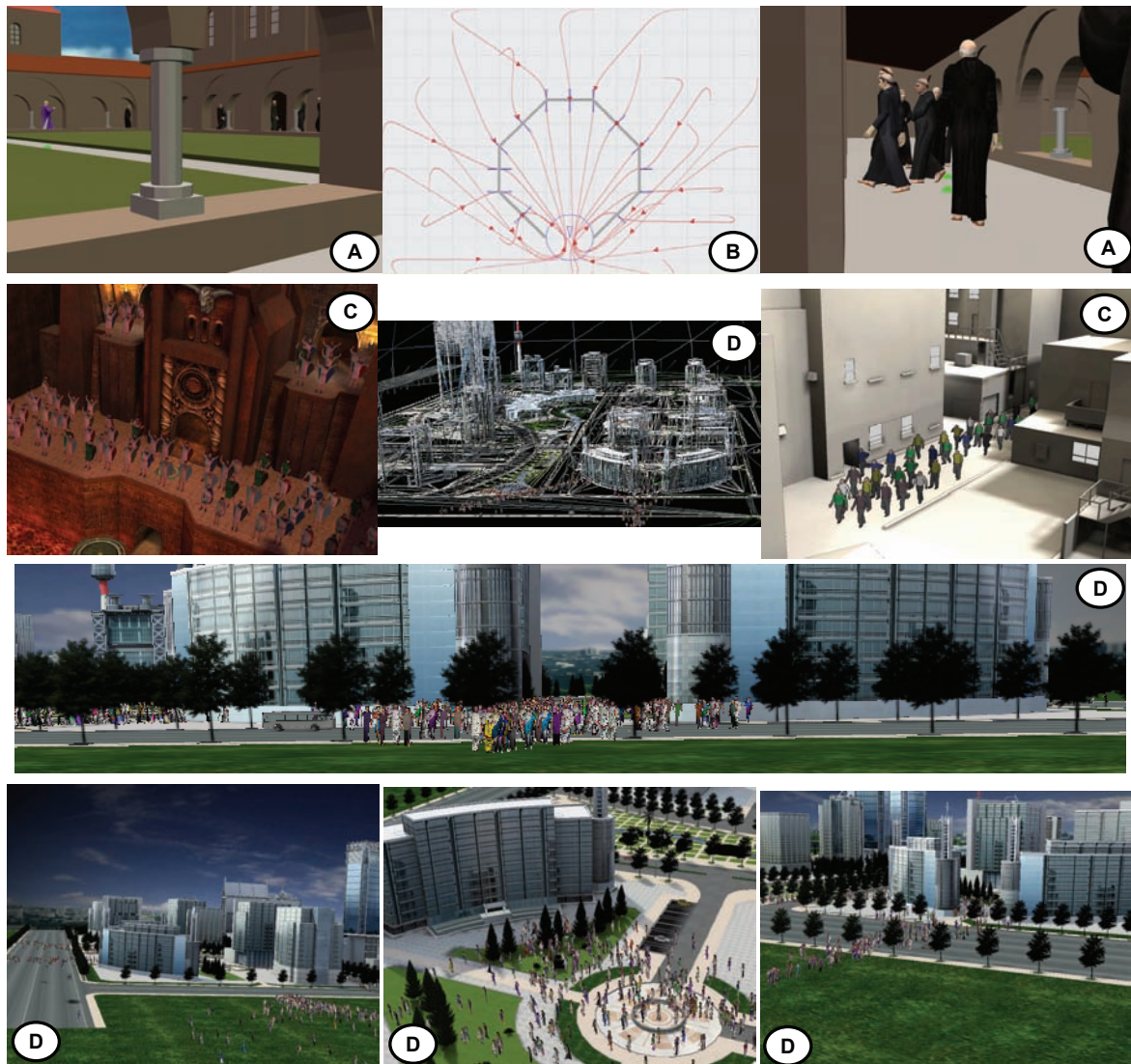


Figure **10.30**: Collection of use cases and examples.

10.9 Conclusion

This chapter gave an overview of different applications developed using the architecture described in this thesis. The wide variety of constraints and scaling factors highlights the system potential to handle many components such as AR devices or crowd simulations. Naturally, the single performance of simulation may be outperformed by specialized system; the flexibility and high potential of code and content reuse greatly facilitate the prototyping of unique simulations. This is important for prototyping simulations before entering into real production as the development costs can be controlled with better efficiency. With today budgets, publishers are interested to keep 3DRTS developments in track by being able to predict investments and minimizing core developments by reusing the same architecture for multiple projects.

Chapter 11

Conclusion

This chapter summarized the results and experience learnt and gives the outlook on further research directions and possible extensions.

11.1 Summary

In this thesis, our goal was to explore systems architectures for 3DRTS. The current scale and hardware trends oblige developers to re-think and adopt stronger engineering practices both for managing the underlined complexity but also to control development costs. In fact, current computers hardware allows generating highly detailed virtual worlds with interactive frame rate. The main difficulty is to incorporate more deeply artists and designers in the development process. By moving from the classical programmers and graphic centric approach toward more flexible and usable platforms will greatly improve both realism and believability of virtual worlds. This trend was once observed at the end of the 2D era, where technical issues were becoming less critical and this have still to emerge for 3DRTS. In the scope of this thesis, the attention was mainly dedicated to promote strong emphasis on design patterns for multitasking data-driven simulations. This implied to ease authoring and manipulations of simulations objects and contents.

11.2 Contributions

The following paragraphs summarize our main results:

Concurrent CBD model: We showed patterns that can be adapted for the specific requirements of multitasking environments and mission-critical software. We proposed a novel architecture that takes its foundations from well-known techniques used in generic software engineering. The idea was to transpose them to the particularities of 3DRTS developments.

Hardware-Oblivious models: we demonstrated that high-level architectures could be applied without penalizing the overall system performance by applying low-level optimizations where it really matters. This includes understanding how computers hardware work and evolve, notably with memory architectures and highly multitasking environments.

Workflow balancing: We highlight that particular platforms may perform some tasks faster and slower at the same time than other. This obliges to incorporate mechanisms to modify the simulation workflow management. In our case, we let developers changing parameters into XML configuration files to adapt the simulation for different platforms. The concrete balancing still require profiling the code using both internal profiler and commercial tools as a validation and optimization phases.

High-Level languages and concurrent model using microthreads: We showed how to extend the system capabilities by embedding scripting languages into our architecture. This gives the opportunity to develop multitasking and dynamic scripts without the common C/C++ issues like memory management or data synchronizations. This increase the development time that is available to experiment with ideas, in a safer environment.

Data-Driven AI engine: we illustrated how to create a generic data-driven AI engine for describing agent's behaviors using LODs. This demonstrates that decoupling the logic and data layers not only increase the system flexibility but also provide the ability to alter a simulation after its deployment. These open opportunities to expose system functionalities to the mod community expending the application lifetime.

Authoring: We illustrated the need to provide more tools and editors for creating and manipulating simulations assets through consistent content creation pipelines, which are required to unleash the system potential. The variety in the tools includes communication layers between DCC tools and the engine as well as GUIs to analyze and edit the content dynamically with WYSIWYG editors. This is an absolute condition for moving systems architectures from their current programmers' centric approach toward more artistic side. In effect, programmers might lack the design styles to create rich content, which are less focus on technological challenges.

Use-Cases: Finally, we demonstrated concrete examples that highlight the framework potential. From AR to crowd simulations, the system versatility clearly illustrate that our approach is functional. Naturally, the scope of this thesis did not allow exposing all the potential of our approach but should already prove the viability of the underlined concepts.

11.3 Discussion and Lessons Learned

The main limitation of our approach is tightly connected to the amount of resources required to develop all the components and tools needed by the different simulations genders. Even so, our approach increase the potential of code and content reuse, the current scale make the maintainability of the framework difficult. Notably, the system requirements include ensuring backward-compatibility with existing libraries such as virtual animations systems relying on the H-ANIM standard. This prevents us to refactor some critical elements. The reasons were twofold: the available resources to redevelop those modules were not available and to allow others researchers to develop within a consistent and stable environment. Another aspect concerns the direct impact of modifications that affect collaborative works. This also limits the potential of major modifications that would break large code base. All these facts help to understand the complexity of developing a generic framework for 3DRTS. In effect, commercial alternatives such as RenderWare [Criterion] of Unreal Technology [Epic-Unreal] dedicate many resources both from financial and humans' aspects that are magnitude higher than what was possible to allocate in the scope of this thesis. While 3DRTS are in many aspects more demanding than other software, in some way they are still ways behind in term of architectures and developments processes. With current hardware, it is possible to sacrifice some performance for better and high-level representations, which would greatly help to promulgate specific design methods. Fortunately, the time where 3DRTS developments were obliged to use one-time programming tricks to keep interactive frame rate is slowly disappearing, due to the added performance throughput and increase complexity. In effect, none application is able to come anywhere close to theoretical numbers of computers hardware, with the notable exception of specialized and confined simulations. As an outcome, the community and the industry as a whole is calling for better methods, that would ease to schedule projects and keep the rising developments costs under control.

11.4 Future Topic

We see several possible directions for future research in the area of 3DRTS architecture systems:

Highly Parallel Design: In our architecture, and as well in other works, the concurrent model is constituted by components that was primary develop as single-threaded components. Some of them were equipped with fine grain parallelisms at a second thought and therefore most of our parallelism is restrained to execute components update in parallel. By developing components with multitasking operations right from the beginning would greatly improve the performance and potential scalability for the next-generation of hardware. This lead to re-think not only the general architecture design as presented in this thesis, but also to re-think how to handle classical 3DRTS problems such as collisions detections in a concurrent world [Ericson2005]. Such approaches would also scale better with different hardware, which could adapt the workflow more accurately upon the number of hardware threads at disposal.

Scalability: In order to increase the scene complexity and to adapt to the hardware capacities, the architecture and more specifically, his rendering component should be equipped with modern techniques for scaling 3D models [Tatarchuk2005]. This include to emphasis on procedurally generated models, which would adapt themselves on the fly. If this can be achieved for 3D content, the scalability is more difficult to handle for other simulation aspects such as behavioral LODs. In our approach, it merely consists to assign importance for each behavioral state, but we still have to generate the states manually. In fact, scalability should be integrated upfront through the usage of meta-data such as the [Collada] initiative.

Variety: The increase complexity and realism of current scenes oblige developers to enforce usage of methods that create a wider variety base on the same content. For instance, in the games franchise Gran Turismo, the modeling of a car was taking 2-3 days and was made of ~250 polygons for the first iteration, and was incrementally increased to one month with a complexity of ~4000 polygons for Gran Turismo 4 [Polyphony2004]. More recent racing games incorporate cars with an average of 96'000 polygons [PGR32005]. This is not a coincidence if procedural methods are investigated [Wright2005]. In our architecture, most of the variety was confined to virtual humans, but many others aspect such as nature scenes [IDV] could benefit from techniques that increase variety with a restricted number of manually generated assets [Tatarchuk2005].

Data-Driven: One primary goal of our architecture was to separate the logic and data layers, allowing non-programmers to use the system more easily. We highlight that C++ core functionalities can be exposed using automatic “*glue code*” generation and how to promote multitasking environments in a safe environment using Microthreads. However, many aspects of our design still require compiling source code. For instance, even so the GUIs benefit of late binding mechanisms using dynamic libraries; they still required to be compiled. Ideally, with the exceptions of the kernel and core functionalities, most elements should become configurable dynamically. The difficulty is always to find the proper balance between performance and flexibility, notably concerning properties or ISOs. In addition, our system lacks of a more complete toolkits and development IDEs such as the one found in RenderWare [Criterion], Unreal Technology [Epic-Unreal] or XNA Studio [Microsoft-XNA].

11.5 Epilogue

The rapid adoption of multitasking processors for computers hardware use for running 3DRTS, such as next-generation of home consoles and PCs platforms evolutions force developers to re-think their algorithms. Approaches that were performing efficiently in older hardware might not perform well on modern architectures. Both the conceptual changes in computer hardware (in-order vs. out-of-order, memory bandwidth...) and highly multitasking processors can explain this. This horizontal increase of performance will lead to the emergence of new simulations that will integrate previously offline techniques that include realistic physics, crowd, or ray tracing. As an outcome, development houses will have to adopt stronger methods for handling this increased complexity to combine the diverse components developed by field specialists. Similar to the movies industry, where the production is spitted among specialized companies (sound, special effects...), the 3DRTS industry need to approach the software practices differently. This thesis was an attempt to standardize high-level architectures and design styles for 3DRTS developments.

Bibliography

- [3DLabs] 3DLabs, (<http://www.3dlabs.com/>).
- [3dvri] 3dvri, "3dvri: OSG modeling and programming service", 3dvri, (<http://www.3dvri.com/>).
- [Abaci2006] Tolga Abaci, "Object manipulation and grasping for virtual humans", EPFL, 2006.
- [Abrahams2004] David Abrahams, Aleksey Gurtovoy, "C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond", 2004.
- [Abrash1994] Michael Abrash, "Zen of Code Optimization: The Ultimate Guide to Writing Software That Pushes PCs to the Limit", Coriolis Group Books, 1994.
- [Acton2005] Mike Acton, "Data Design for Consoles", GDC, 2005.
- [Adams-FP2005] Ernest W. Adams-FP, "Emerging Issues in Game Design", FuturePlay Conference, 2005.
- [Adams2004] Ernest W. Adams, "The Future of Computer Entertainment", Advances in Computer Entertainment Conference, Singapore, 2004.
- [Adams2003] Jim Adams, "Advanced Animation with DirectX", Premier Press, 2003.
- [Adobe] Adobe, "Photoshop CS", Adobe Software, (<http://www.adobe.de/products/photoshop/main.html>).
- [Adolph2004] Steve Adolph, "Reuse And Staying in the Business", Gamasutra, 2004.
- [Advanced_Interfaces_Group1999] Advanced_Interfaces_Group, "MAVERIK - a VR micro kernel", The Advanced Interfaces Group, (<http://www.gnu.org/software/maverik/maverik.html>), 1999.
- [Adya2002] A. Adya, J. Howell, M. Theimer, W.J. Bolosky, J.R. Doucer, "Cooperative Task Management without Manual Stack Management", In Proceedings of USENIX, Annual Technical Conference, Monterey, California, 2002.
- [Agarwal2003] Pankaj K. Agarwal, Lars Arge, Andrew Danner, Bryan Holland-Minkley, "Cache-Oblivious Data Structures for Orthogonal Range Searching", 2003.
- [Ageia] Ageia, "PhysX", Ageia, (<http://www.ageia.com/>).
- [Ageia2006] Ageia, "Advanced Gaming Physics: Defining the New Reality in PC Hardware", White Paper, Ageia, 2006.
- [Aguaviva2005] Raul Aguaviva, Jeff Kiel, "Performance Tools", GDC, 2005.
- [Akenine-Moller2002] Thomas Akenine-Moller, Eric Haines, "Real-Time Rendering, A.K. Peters", A.K. Peters Ltd, 2002.
- [Alexander2002] Bob Alexander, "An Architecture Based on Load Balancing", in AI Game Programming Wisdom, Charles River Media, 2002.
- [Alexander1977] Christopher Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-Kingand, S. Angel "A Pattern Language", New York, Oxford University Press, 1977.
- [Alexandrescu] Andrei Alexandrescu, "Loki: A C++ library of designs, containing flexible implementations of common design patterns and idioms." (<http://sourceforge.net/projects/loki-lib/>).
- [Alexandrescu2001] Andrei Alexandrescu, "Modern C++ Design: Generic Programming and Design Patterns Applied", Addison Wesley Professional, 2001.
- [Alias] Alias, "Maya", Alias, (<http://www.alias.com/eng/index.shtml>).
- [Alias2005] Alias, "Learning Maya 7 Book Bundle", Alias, 2005.
- [Allard2004] J. Allard, "Microsoft Keynote: Unveils XNA", GDC, 2004.
- [Alves2004] Carina Alves, "Analysing the Tradeoffs Among Requirements, Architecture and COTS Components", Centro de Informatica, Universidade Federal de Pernambuco Recife, Pernambuco, 2004.
- [Ambler2002] Scott W. Ambler, Ron Jeffries, "Agile Modeling: Effective Practices for Extreme Programming and the Unified Process", 2002.
- [AMD-CA] AMD-CA, "AMD CodeAnalyst: Performance Analyzer for Windows", AMD, (http://www.amd.com.cn/chcn/Processors/DevelopWithAMD/0_30_2252_3604.00-1.html).
- [AMD] AMD, "Advanced Micro Devices", (<http://www.amd.com>).
- [Amdahl1967] G.M. Amdahl, "Validity of the single-processor approach to achieving large scale computing capabilities", AFIPS Conference Proceedings, AFIPS Press, 30, pp. 483-485, 1967.
- [Ancel2005] Michel Ancel, "King Kong: The game", Ubi Soft, (<http://www.kingkonggame.com/>), 2005.
- [Anderson2003] Eike F. Anderson, "Playing Smart - Artificial Intelligence in Computer Games", zfx Conference 2003.
- [Andrews2005] Jeff Andrews, "Preparing for Hyper-Threading Technology and Dual-Core Technology", White Paper, Intel, 2005.
- [Anvik2002] John Anvik, Steve MacDonald, Duane Szafron, Jonathan and Steven Bromling Schaeffer, Kai Tan, "Generating Parallel Programs from the Wavefront Design Pattern", in Proceedings of the 7th International Workshop on High-Level Programming Models and Supportive Environments, 2002.
- [Apache-Ant] Apache-Ant, "Apache Ant: a Java-based build tool", (<http://ant.apache.org/>).
- [Appleton2000] Brad Appleton, "Patterns and Software: Essential Concepts and Terminology", (<http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html>), 2000.
- [Arafa2002] Yasmine Arafa, Kaveh Kamyab, Sumedha Kshirsagar, Anthony Guye-Vuillème Anthony Magnenat-Thalmann Nadia, Daniel Thalmann "Two Approach to Scripting Character Animation", Autonomous Agents & Multi-Agent Systems (AAMAS), 2002.
- [Arnaud1999] R. Arnaud, M. Jones, "Innovative Software Architecture for Real-Time Image Generation", Interservice/Industry Training Systems and Equipment Conference (I/ITSEC) Conference Ontario, Florida 1999.

- [Artistic-Style] Artistic-Style, "Artistic Style", (<http://sourceforge.net/projects/astyle/>).
- [Asaduzzaman2003] Abu Asaduzzaman, "Performance Evaluation of Embedded Systems", 2003.
- [Astels2003] David Astels, "Test Driven Development: A Practical Guide ", Prentice Hall PTR, 2003.
- [Astle2006] Dave Astle, "More OpenGL Game Programming: Chapter 12 - Scene Management", Thomson Course Technology, 2006.
- [ATI-OPT2006] ATI-OPT, "ATI OpenGL Programing and Optimization Guide", ATI, 2006.
- [ATI-R2VB2006] ATI-R2VB, "R2VB Skinning animation", ATI, 2006.
- [ATI-RM] ATI-RM, "RenderMonkey", ATI, (<http://www.ati.com/developer/techpapers.html#gdc06>).
- [Atkin2001] M.S. Atkin, D.L. Westbrook, "Panel Discussion: Collaboration Between Academia and Industry: A Case Study", Working Notes of AAAI Spring Symposium, 5-6, 2001.
- [Attardi1998] Giuseppe Attardi, Tito Flagella, Pietro Iglo, "A customizable memory management framework for C++", 1998.
- [Aue2005] Anthony Aue, "Improving Performace With Custom Allocators for STL", in Dr. Dobb's Journal, CMP, September, 2005.
- [Avid-Alienbrain] Avid-Alienbrain, "Alienbrain Studio", Avid, (<http://www.alienbrain.com/>).
- [Backman] Anders Backman, "osgAL", (www.sf.net/projects/osgal).
- [Baekkelung2006] Christian Baekkelung, "Academic AI Research and Relations with the Game Industry", in AI Game Programming Widsom 3, Charles River Media, 2006.
- [Bagley2005] Doug Bagley, "The Computer Language Shootout Benchmarks", (<http://shootout.alioth.debian.org/>), 2005.
- [Balfour2006] Michael Balfour, Daniel Martin, "Sim, Render, Repeat – An Analysis of Game Loop Architectures", GDC, 2006.
- [Bar-Zeev2003] Avi Bar-Zeev, "Scenegraps: Past, Present and Future", 2003.
- [Bartolomeo2003] Dave Bartolomeo, "Optimizing Code with Visual Studio .NET", Meltdown, 2003.
- [Bartz2001] Dirk et al. Bartz, "Jupiter: A ToolKit for Interactive Large Model Visualization", IEEE Symposium on parallel and large data visualization anf graphics proceedings, IEEE, 2001.
- [Bass2003] Len Bass, Paul Clements, Rick Kazman, "Software Architecture in Practice, Second Edition", Addison-Wesley, 2003.
- [BBC2005] BBC, "Fans hit back in GTA sex storm", BBC News, (<http://news.bbc.co.uk/1/hi/technology/4735603.stm>), 2005.
- [Beazley1999] David M. Beazley, "Python: Essential Reference", New Riders, 1999.
- [Beck2002] Kent Beck, "Test Driven Development: By Example", Addison-Wesley Professional, 2002.
- [Beck2004] Kent Beck, Cynthia Andres, "Extreme Programming Explained : Embrace Change - 2nd Edition", Addison-Wesley, 2004.
- [Bender2000] M.A. Bender, E. Demaine, M. Farach-Colton, "Cache-oblivious B-trees", In Proc. 41st Ann. Symp. on Foundations of Computer Science, pages 399-409, IEEE Computer Society Press, 2000.
- [Berger2000] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, Wilson Paul R., "Hoard: A Scalable Memory Allocator for Multithreaded Applications", 2000.
- [Berger2001] Emery D. Berger, Benjamin G. Zorn, Kathryn S. McKinley, "Reconsidering Custom Memory Allocation", 2001.
- [Berger2002] Lee Berger, "Scripting: Overview and Code Generation", in AI Game Programming Widsom, Charles River Media, 2002.
- [Berstein1996] Philip A. Berstein, "Middleware: A Model for Distributed System Services", ACM, 39, pp 86-98, 1996.
- [Berthel2003] Wes Berthel, "White Paper: Sort-First Distributer Memory Parallel Visualization and Rendering with OpenRM Scene Graph and Chromium", R3vis Corporation, 2003.
- [Bethke2003] Erik Bethke, "Game Development and Production", Wordware Publishing, Inc., 2003.
- [Bhatti2003] S. Bhatti, "Memory Architecture", UCL Computer Science, 2003.
- [Bignami] Luigi Bignami, "UML Pad", (<http://web.tiscali.it/ggbhome/umlpad/umlpad.htm>).
- [BigWorld] BigWorld: a complete solution for developers of Massively Multiplayer Online Games (MMOGs)." (<http://www.bigworldtech.com/>).
- [Bilas-ISL2001] Scott Bilas-ISL, "Is Still Loading? Designing an Efficient File System", GDC 2001, 2001.
- [Bilas2000] Scott Bilas, "An Automatic Singleton Utility", in Game Programming Gems, Charles River Media, 2000.
- [Bilas2001] Scott Bilas, "Automatic Function Exporting for Networking and Scripting ", GDC, 2001.
- [Bilas2002] Scott Bilas, "A Data-Driven Game Object System", Gas Powered Games, 2002.
- [Bilas2003] Scott Bilas, "The Continuous World of Dungeon Siege", GDC, 2003.
- [Bin1998] Li Bin, "Software Reuse", (<http://sern.ucalgary.ca/courses/seng/693/W98/lib/reuse.htm>), 1998.
- [Binstock2003] Andrew Binstock, "Multithreading, Hyper-threading, Multiprocessing: Now, what's the difference?" Pacific Data Works, 2003.
- [BioGraphics] BioGraphics, "AI Implant", BioGraphics, (<http://www.biographictech.com/>).
- [Bionathics] Bionathics, "REALnat", Bionatics, (<http://www.bionatics.com/>).
- [Bishop2005] Christopher M. Bishop, "Machines that Learn: Adaptative Computing in the 21st Century", Lovelace, 2005.
- [Bjorke2005] Kevin Bjorke, "Let's Get Small", Nvidia, 2005.
- [Blow2003] Jonathan Blow, "Profiler", GDMag January, 2003.
- [Blow2004] Jonathan Blow, "Game Development: Harder Than You Think ", ACM Queue, 2004.
- [Blunden2002] Bill Blunden, "Memory Management : Algorithms and Implementations In C/C++", Wordware Publishing Inc., 2002.
- [Blythe2000] David Blythe, "Migrating to an Object-Oriented Graphics API", Siggraph, 2000.
- [BMBF] BMBF, "OpenSG", (<http://www.opensg.org>).

- [Boehm1984] B.W. Boehm, M. Pendo, A. Pyster, E.D. and William Stuckle, R.D., "An Environment for Improving Software Productivity", IEEE Computer, 1984.
- [Boer-OOP2000] James Boer-OOP, "Object-Oriented Programming and Design Techniques", in Game Programming Gems, Charles River Media, 2000.
- [Boer2004] James Boer, "An HTML-Based Logging and Debugging System", in Games. Programming Gems 4, Charles River Media, 2004.
- [Boer2000] John Boer, "Resource and Memory Management", in Game Programming Gems, Charles River Media, 2000.
- [Bogojevic2003] Sladjan Bogojevic, Mohsen Kazemzadeh, "The Architecture of Massive Multiplayer Online Games", Department of Computer Science, Lund Institute of Technology, Sweden, 2003.
- [Bogost2006] Ian Bogost, "Unit Operations : An Approach to Videogame Criticism", The MIT Press, 2006.
- [Booch2004] Grady Booch, "Handbook of software architecture: Software Architecture", IBM Software Group, Rational software, (Booch), 2004.
- [Booch2005] Grady Booch, "The Complexity of Programming Models", IBM Rational, 2005.
- [Booch2006] Grady Booch, "Best Practices for Game Development", GDC, 2006.
- [Booch] Grady Booch, "Handbook of Software Architecture", (http://www.booch.com/architecture/patterns.jsp?view=kind_name%20catalog).
- [Boost-Test-Library] Boost-Test-Library, "Boost Test Library", (<http://boost.org/libs/test/doc/index.html>).
- [Boost] Boost, "Boost C++ Libraries", (<http://www.boost.org>).
- [Borovikov2005] Igor Borovikov, "An Orwellian Approach to AI Architecture", GDC, 2005.
- [Boulie2004] Ronan Boulie, Branislav Ulicny, Daniel Thalmann, "Versatile Walk Engine", Journal Of Game Development, Charles River Media, vol. 1, pp. 29-54, 2004.
- [Bourg2001] David Bourg, "Physics for Game Developers", O'Reilly Media, 2001.
- [Bouvier2005] Albert-Jan Bouvier, "Embedding Lua into LabVIEW", Lua: Workshop, 2005.
- [Boyd2003] Chas Boyd, "Future Features", Meltdown, 2003.
- [Brockington2002] Mark Brockington, "Level-Of-Detail AI for a Large Role-Playing Game", in AI Game Programming Wisdom, Charles River Media, 2002.
- [Brodal2003] Storting Brodal, Gerth, "Cache Oblivious Searching and Sorting", 2003.
- [Brogan2000] David C. Brogan, Jessica K. Hodgins, "Simulation Level of Detail for Multiagent Control", 2000.
- [Brooks Jr1995] Fred P. Brooks Jr, "The Mythical Man-Month: Essays on Software Engineering", Addison-Wesley Professional, 1995.
- [Brooks1999] Rodney A. Brooks, "Cambrian Intelligence: The Early History of the New AI", The MIT Press, 1999.
- [Brown2005] Jeffrey Brown, "Application-customized CPU designThe Microsoft Xbox 360 CPU story", Fall Processor Forum, 2005.
- [Brown1999] William Brown, Raphael Malveau, Hays McCormick, Thomas and Thomas Mowbray, Scott W., "AntiPatterns", (http://www.antipatterns.com/arch_cat.htm), 1999.
- [Brownlow2004] Martin Brownlow, "Game Programming Golden Rules", Charles River Media, 2004.
- [Bruckschlegel2005] Thomas Bruckschlegel, "Micro Benchmarking C++, C#, and Java", in C/C++ User Journal, CMP, vol. 23, 2005.
- [Buchanan2005] Warrick Buchanan, "A Generic Component Library", in Programming Gems 5, Charles River Media, 2005.
- [Buck2004] Ian Buck, Tim Purcell, "A Toolkit for Computation on GPUs", in GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics, Addison Wesley, 2004.
- [Buckland2002] Mat Buckland, "AI Technique for Game Programming", Premier Press, 2002.
- [Buckland2005] Mat Buckland, "Programming Game AI by Example", Wordware Publishing Inc., 2005.
- [Bungie2005] Bungie, "Research Game Development: Halo 2", Bungie, 2005.
- [Burger1996] Doug Burger, James R. Goodman, K. Alain, "Memory bandwidth limitations of future microprocessors ", Proceedings of the 23rd annual international symposium on Computer architecture, ACM, 78-89 1996.
- [Buschmann1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter and Stal Sommerlad, Michael and Sommerlad, Peter and Stal, Michael, "Pattern-Oriented Software Architecture, Volume 1: A System of Patterns", John Wiley & Sons, 1996.
- [Cal3d] Cal3d, "Cal3d: Character Animation Library ", (<http://cal3d.sourceforge.net/>).
- [Caldiera1991] G. Caldiera, V.R. Basili, "Identifying and Qualifying Reusable Software Components", In IEEE Computer, pp. 61-70, 1991.
- [Calver2005] Dean Calver, "The Sharp Evil of the Next Generation: PC to PS3", GDCE, 2005.
- [Calvert1996] David Calvert, "Software Architectural Styles", (<http://hebb.cis.uoguelph.ca/~dave/27320/new/architec.html>), 1996.
- [Capps2000] Michael Capps, Don McGregor, Don Brutzman, Michael Zyda, "NPSNET-V: A New Beginning for Dynamically Extensible Virtual Environments ", IEEE Comput. Graph. Appl., 2000.
- [Carey1997] Rikk Carey, G. Bell, "The Annotated VRML 2.0 Reference Manual", Addison-Wesley, 1997.
- [Carrera] Andres Carrera, "oFusion: a WYSWYG toolset for developing content for the Ogre3D engine", ACE Studios, (<http://ofusion.inocentric.com/index.html>).
- [Carter2004] Ben Carter, "The Game Asset Pipeline", Charles River Media, 2004.
- [Carter2002] David Carter, "Introducing PS2 to PC Programmers", AGDC, 2002.
- [Carter2001] Simon Carter, "Managing AI with Micro-Threads", in Game Programming Gems 2, Charles River Media, 265-272, 2001.
- [Carucci2005] Francesco Carucci, "Inside Geometry Instancing", in GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation Addison Wesley, 2005.
- [Carver2005] Richard H. Carver, Kuo-Chung Tai, "Modern Multithreading : Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs", Wiley-Interscience, 2005.

- [Casey2005] Wardynski Casey, "America's Army: The Evolution of a Successful Serious Game", Serious Games Submit, Washington, 2005.
- [Cebenoyan2004] Cem Cebenoyan, "Graphics Pipeline Performance", GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics, Addison Wesley, 2004.
- [Cg] Cg, "The Cg Shader Programming Language", (http://developer.nvidia.com/page/cg_main.html).
- [Champandard2003] Alex Champandard, "AI Game Development, Synthetic Creatures with Learning and Reactive Behaviors", New Rider, 2003.
- [Champoux2006] Bruno Champoux, Nicolas Fleury, "Feeding the Monster: Advanced Data Packaging for Consoles", GDC, 2006.
- [Chapling2005] Heather Chapling, Aaron Ruby, "Smartbomb: The Quest for Art, Entertainment, and Big Bucks in the Videogame Revolution", Algonquin Books of Chapel Hill, 2005.
- [Charlebois2005] Pierre-Olivier Charlebois, "Graphical Representation of Sound for the "Soundscape" Immersive Environment using the PD-Gem (OpenGL) Framework", McGill University, 2005.
- [Chatelaine2003] Jeremy Chatelaine, "Enabling Data Driven Tuning Via Existing Tools", GDC, 2003.
- [Chenney2002] Stephen Chenney, "Simulation Level-Of-Detail", University of Wisconsin, 2002.
- [Chilimbi1999] Trishul Chilimbi, Bob Davidson, James Larus, "Cache-conscious Structure Definition", Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI 99), pp. 13-24, 1999.
- [Cho2002] Namkyu Cho, "The Development Process", Component Based Development, CBD Workshop, 2002.
- [Choi2000] Sung-Eun Choi, E. Chirstopher Lewis, "A Study of Common Pitfalls in Simple Multi-Threaded Programs", Proceedings of the thirty-first SIGCSE technical symposium on Computer science education, 2000.
- [Church2004] Doug Church, "Object Systems", GDC, 2004.
- [Ciger2005] Jan Ciger, "Collaboration with Agents in VR environments", Phd Thesis, EPFL, 2005.
- [Clinton2003] Keith Clinton, "From the Ground: Creating a Core Technology Group", Gamasutra, (http://www.gamasutra.com/features/20030801/keith_01.shtml), 2003.
- [CMake] CMake, "CMake: Cross-platform Make", (<http://www.cmake.org/HTML/Index.html>).
- [Collada] Collada, "COLLADA: An open Digital Asset Exchange Schema for the interactive 3D industry." (<https://collada.org/>).
- [Compuware] Compuware, "DevPartner Studio", Compuware, (www.compuware.com/products/devpartner/default.htm).
- [Conway1963] D. Conway, "Design of a Separable Transition-Diagram Compiler", CACM, 1963.
- [Corel] Corel, "Paint Shop Pro", Corel, (<http://www.corel.com>).
- [Cormen2001] Thomas Cormen, "Introduction to Algorithms, 2nd Edition", MIT Press, 2001.
- [Cosmo3D1998] Cosmo3D, "Cosmo3D Programmer's Guide", Silicon Graphics Inc., 1998.
- [Costa2004] S. Costa, "Game Engineering for a Multiprocessor architecture", Msc Dissertation Computer Graphic Technology, Jonh Moores University, 2004.
- [CPPUnit] CPPUnit, "CppUnit: a C++ unit testing framework inspired from JUnit", (<http://cppunit.sourceforge.net/>).
- [Creative-OpenAL] Creative-OpenAL, "OpenAL", Creative Labs, (<http://www.openal.org/>).
- [Criterion] Criterion, "RenderWare", Electronic Arts.
- [Crystal_Space] Crystal_Space, "Crystal Space", (<http://www.crystalspace3d.org/>).
- [CVS] CVS, "CVS: Concurrent Versions System", (<http://www.nongnu.org/cvs/>).
- [CXX] CXX, "A library for writing Python extensions in C++".
- [CXXTest] CXXTest, "CxxTest: a JUnit/CppUnit/xUnit-like framework for C++", (<http://cxxtest.sourceforge.net/>).
- [Dachselt2002] R. Dachselt, M. Hinz, K. Meissner, "CONTRIGA: An XML-Based Architecture for Component-Oriented 3D Applications", ACM Web3D Symposium, 2002.
- [Daglow2005] Don L. Daglow, "Next Gen Game Production: What Changes in the Process?", GDC Europe, 2005.
- [Dahl1997] Ole-Johan Dahl, Kristen Nygaard, "How Object-Oriented Programming Started", Wexeblat, 1997.
- [Dahlgren1999] Fredrik Dahlgren, Josep Torrellas, "Cache-Only Memory Architectures", IEEE, 1999.
- [Dalmau2002] Daniel Sanchez-Crespo Dalmau, "Its a Complex (Game) World", Gamasutra, 2002.
- [Dalmau2003] Daniel Sanchez-Crespo Dalmau, "Core Techniques And Algorithms in Game Programming", New Riders Publishing, 2003.
- [Dalton2001] Peter Dalton, "A Drop-in Debug Memory Manager", Game Programming Gems 2, Charles River Media, 2001.
- [Dandashi2001] Fatma Dandashi, David C. Rine, "A Method for Assecing the Reusability of Object-Oriented Code Using a Validated Set Of Automated Measurements", Mitre, 2001.
- [Darovsky2005] Alexander Darovsky, "FSME", (<http://fsme.sourceforge.net/>), 2005.
- [Davies-MT2005] Leigh Davies-MT, "Multi-Threading Games For Performance", GDCE, 2005.
- [Davies2005] Leigh Davies, "Real-life case studies in multi-threading game applications for performance", GDC, 2005.
- [Davies2006] Leigh Davies, "Multi-Threading Games For Performance", GDC, 2006.
- [Dawson2001] Bruce Dawson, "Micro-Threads for Game Object AI", in Game Programming Gems 2, Charles River Media, 2001.
- [Dawson2002] Bruce Dawson, "Game Scripting in Python", GDC, 2002.
- [Dawson2006] Bruce Dawson, Chuck Walbourn, "Coding for Multiple Cores", GDC, 2006.
- [de Figueiredo2006] Luiz Henrique de Figueiredo, Waldemar Celes, "Programming Advanced Control Mechanisms with Lua Coroutines", in Game Programming Gems 6, Charles River Media, 2006.
- [De Gelas-PI2005] Johan De Gelas-PI, "The Quest for More Processing Power, Part One: "Is the single core CPU doomed?" Anandtech, 2005.
- [De Gelas-PH2005] Johan De Gelas-PH, "The Quest for More Processing Power, Part Two: "Multi-core and multi-threaded gaming", Anandtech, 2005.

- [de Gentile-Williams2005] Marc de Gentile-Williams, Nick Gibson, Ben Keen, Nick Parker, "Games Software Publishing: Strategies for market success", 2005.
- [de Heras-VAST2005] Pablo de Heras-VAST, Sebastien Schertenleib, Jonathan Maim, Daniel Thalmann, "Reviving the Roman Odeon of Aphrodisias: Dynamic Animation and Variety Control of Crowds in Virtual Heritage", VAST, 2005.
- [de Heras-VSMM2005] Pablo de Heras-VSMM, Sebastien Schertenleib, Jonathan Maim, Damien Maupu, Daniel Thalmann, "Real-Time Shader Rendering for Crowd in Virtual Heritage", VSMM, 2005.
- [De Margheriti2005] John De Margheriti, "Using BigWorld Technology to Create your First MMOG", GDCE, 2005.
- [de Sevin2005] Etienne de Sevin, "A Motivational Model of Action Selection for Virtual Humans", Computer Graphics International (CGI), IEEE Computer Society Press, IEEE Computer Society Press, New York, 2005.
- [de Sevin2006] Etienne de Sevin, "An Action Selection Architecture for Autonomous Virtual Humans in Persistent Worlds", EPFL, 2006.
- [Deloura2005] Marc Deloura, "CELL: A New Platform for Digital Entertainment", GDC, 2005.
- [Delta3D_Dev_Team] Delta3D_Dev_Team, "Delta3D", (<http://www.delta3d.org/>).
- [Demachy2003] T. Demachy, "Extreme Game Development: Right on Time, Every Time", Gamasutra, 2003.
- [Demeter] Demeter, "Demeter Terrain Engine", Demeter Consulting Group, (<http://www.terrainengine.com/>).
- [Denman2003] Stuart Denman, "Highly Detailed Continuous Worlds: Streaming Game Resources From Slow Media", GDC, 2003.
- [Objects by Design] Objects by Design, "Objects by Design: a site dedicated to bringing you valuable information about the world of object-oriented design and programming", Objects by Design, (http://www.objectsbydesign.com/tools/umltools_byPlatform.html).
- [Devir2002] Zwi Devir, Ronen Zohar, "Optimized Matrix Library for use with the Intel Pentium 4 Processor's Streaming SIMD Extensions (SSE2)", Intel Developer Service, (<http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/code/20460.htm>), 2002.
- [Dewhurst2002] Stephen C. Dewhurst, "C++ Gotchas: Avoiding Common Problems in Coding and Design", Addison Wesley Professional, 2002.
- [Dewhurst2003] Stephen C. Dewhurst, "C++ Memory and Resource Management", Addison Wesley Professional, 2003.
- [Dhupelia2005] Shekhar V. Dhupelia, "The Shift to Middleware: Technological Forecast for Game Development", in Secrets of the Game Business, Second Edition, Charles River Media, 2005.
- [Dietrich2003] Sim Dietrich, "Modern Graphics Engine Design", Nvidia, 2003.
- [Digimation-BonePro] Digimation-BonePro, "Bone Pro 3", Digimation, (<http://www.digimation.com/home/>).
- [Dijkstra1965] Edsger W. Dijkstra, "Solution of a Problem in Concurrent Programming Control", ACM, 8, 1965.
- [Dijkstra1989] Edsger W. Dijkstra, "On the Cruelty of Really Teaching Computer Science", Communications of the ACM, vol. 32, 1989.
- [Ding2004] Chen Ding, "Data Layout Optimizations", Computer Organization, Rochester, 2004.
- [DirectX] DirectX, "DirectX: Software Development Kit", Microsoft, (<http://www.microsoft.com/directx>).
- [DirectX_Dev_Team] DirectX_Dev_Team, "XACT", Microsoft, (<http://msdn.microsoft.com/directx/xact/>).
- [Discreet] Discreet, "3D Studio MAX", Autodesk, (<http://www.discreet.com>).
- [Doggett2005] Michael Doggett, "Xenos: XBOX360 GPU", Eurographics, 2005.
- [Doherty2003] Michael Doherty, "A Software Architecture for Games", in University of the Pacific Department of Computer Science Research and Projects Journal (RAPJ), vol. 1, 2003.
- [Döllner2000] Jürgen Döllner, Klaus Hinrichs, "A Generalized Scene Graph", Universität Munster, 2000.
- [Domine-GPUJ2004] Sebastien Domine-GPUJ, "Performance Tools and Performance Analysis Techniques", GPU Jackpot, 2004.
- [Dominic2001] François Dominic, "A Game Entity Factory", in Game Programming Gems 2, Charles River Media, 2001.
- [Douglass1999] B.P. Douglass, "Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns", Addison-Wesley, 1999.
- [Dowd1996] P. Dowd, J. Perreault, J. Chu, D. C. Hoffmeister, R. Minnich, D. Burns, F. Hady, Y-J. Chen, M. Dagenais, D. Stone, "LIGHTNING Network and System Architecture", Journal of Lightwave Technology, vol. 14, pp. 1371-1387, 1996.
- [Driesen1996] Karel Driesen, Urs Hölzle, "The direct cost of virtual function calls in C++", OOPSLA, ACM Press, vol. 31, pp. 306-323, 1996.
- [Dudash2006] Bryan Dudash, "DX10, Batching, and Performance Considerations", GDC, 2006.
- [Duffy2004] R. Duffy, "Software Architecture", (<http://members.aol.com/rduffy4187/report.html>), 2004.
- [Duquette2005] Patrick Duquette, "A Real-Time Remote Debug Message Logger", in Programming Gems 5, Charles River Media, 2005.
- [Duran2003] Alex Duran, "Building Object Systems: Features, Tradeoffs, and Pitfalls", GDC, 2003.
- [Dysband2003] Eric Dysband, "AI Middleware: Getting into Character Conclusion", Gamasutra, (http://www.gamasutra.com/features/20030725/dybsand_01.shtml), 2003.
- [Eberly2000] David H. Eberly, "3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics", San Francisco, Morgan Kaufmann Publishers Inc, (<http://www.magic-software.com/>), 2000.
- [Eberly2003] David H. Eberly, "Game Physics", Morgan Kaufmann, 2003.
- [Eberly2004] David H. Eberly, "3D Game Engine Architecture, First Edition : Engineering Real-Time Applications with Wild Magic", Morgan Kaufmann, 2004.
- [Eckel2001] Bruce Eckel, "Thinking in Patterns: Problem-Solving Techniques using Java", MindView, 2001.
- [Egger1995] Susan Egger, Hank Levy, "Simultaneous Multithreading Project", Washington University, (<http://www.cs.washington.edu/research/smt/>), 1995.

- [EiBelle2006] Mike EiBelle, "GPU Performance of DirectX 9 Per-Fragment Operations Revisited", in ShaderX4: Advanced Rendering Techniques, Charles Rivers Media, 2006.
- [El Rhalibi2005] Abdenmour El Rhalibi, Steve Costa, David England, "Game Engineering for a Multiprocessor Architecture", DiGRA, 2005.
- [Emergent] Emergent, "Gamebryo", Emergent Game Technologies, (<http://www.emergentgametech.com/index.php?source=gamebryo>).
- [Epic-Unreal] Epic-Unreal, "Unreal Technology", Epic Games, (<http://www.unrealtechnology.com/flash/technology/ue30.shtml>).
- [Epic-UnrealEd] Epic-UnrealEd, "UnrealEd", Epic Games, (<http://udn.epicgames.com/Two/UnrealEd>).
- [EPIC-UnrealGame1998] EPIC-UnrealGame, "Unreal", Epic Games, 1998.
- [EPOCH] EPOCH, "Excellence in Processing Open Cultural Heritage", European Network of Excellence, IST-2002-507382 funded by the European Commission under the Community's Sixth Framework Programme., (<http://www.epoch-net.org/>).
- [Erato] Erato, "ERATO: Identification Evaluation and Revival of the Acoustical heritage of ancient Theatres and Odea", ECO-MED.
- [Ericson-NRGC2005] Christopher Ericson-NRGC, "Numerical Robustness for Geometric Calculations", GDC, 2005.
- [Ericson2003] Christer Ericson, "Memory Optimization", Santa Monica, Sony Computer Entertainment, 2003.
- [Ericson2005] Christer Ericson, "Real-Time Collision Detection", Morgan Kaufmann Publishers, 2005.
- [Erleben2005] Kenney Erleben, Jon Sporring, Knud Henriksen, Henrik Dohlmann, "Physics Based Animation", Charles River Media, 2005.
- [Esmerado2001] J. Esmerado, "A Model of Interaction between Virtual Humans and Objects: Application to Virtual Musicians", Phd Thesis, EPFL, 2001.
- [Etherton2004] David Etherton, "Designing and Maintaining Large Cross-Platform Libraries", in Game Programming Gems 4, Charles River Media, 2004.
- [Evans2002] Richard Evans, "Varieties of Learning", in AI Game Programming Wisdom, Charles River Media, 2002.
- [Evertt2001] Jeff Evertt, "A Built-in Game Profiling Method", in Game Programming Gems 2, Charles River Media, 2001.
- [Fagerberg2002] Brodal R. Fagerberg, R. Jacob, "Cache oblivious search trees via binary trees of small height", ACM Press, Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms, 2002.
- [Fairclough2001] C. Fairclough, "Research Directions for AI in Computer Games", Technical Report TCD-CS-2001-29, Trinity College Dublin, 2001.
- [Fear] Fear, "Flexible Embodied Animat Architecture", (<http://fear.sourceforge.net>).
- [Fernando2003] Randima Fernando, Mark J. Kilgard, "The Cg Tutorial", Addison-Wesley, 2003.
- [Filion2005] Dominic Filion, "Using Template for Reflection in C++", in Programming Gems 5, Charles River Media, 2005.
- [Finney2005] Kenneth Finney, "Advanced 3D Game Programming All in One", Course Technology PTR, 2005.
- [Firelight] Firelight, "FMOD Ex", Firelight Technologies, (<http://www.fmod.org/>).
- [Firestone2005] Mary Firestone, "Weird Careers in Science: Computer Game Developer", Chelsea House Publishing, 2005.
- [Fleisz2006] Martin Fleisz, "Spatial Partitioning Using an Adaptive Binary Tree", in Game Programming Gems 6, Charles River Media, 2006.
- [Flood2003] K. Flood, "Game Unified Process", 2003.
- [Flynn1966] M.J. Flynn, "Very High-Speed Computing Systems", Proceeding of the IEEE, vol. 54, 1966.
- [Flynt2005] John P. Flynt, Omar Salem, "Software Engineering for Game Developers", Course Technology PTR, 2005.
- [Foltz2003] Mark A. Foltz, "Dr. Jones: A Software Design Explorer's Crystal Ball", MIT, 2003.
- [Fowler-UML1999] Martin Fowler-UML, Kendall Scott, "UML Distilled: A Brief Guide to the Standard Object Modeling Language", Addison Wesley, 1999.
- [Fowler1999] Martin Fowler, Kent Beck, John Brant, William and Roberts Opdyke, Don "Refactoring: Improving the Design of Existing Code", Addison-Wesley, 1999.
- [Freeman-Hargis2006] James Freeman-Hargis, "Using STL and Patterns for Game AI", in AI Game Programming Wisdom 3, Charles River Media, 2006.
- [Frigo1999] M. Frigo, C.E. Leiserson, H. Prokop, S. Ramachandran, "Cache-Oblivious Algorithms", In Proc. 40th Annual Symposium on Foundations of Computer Science (FOCS), pp. 285-297, 1999.
- [Frisbie] Phil Frisbie, "HawkNL: Hawk Network Library", Hawk Software, (<http://www.hawksoft.com/hawknl/>).
- [Fristorm2004] J. Fristorm, "Manager in a Strange Land: Reuse and Replace", Gamasutra, 2004.
- [Fu2002] Dan Fu, Ryan Houlette, "Putting AI in Entertainment: An AI Authoring Tool for Simulation and Games", In IEEE Intelligent Systems, pp 81-84., 2002.
- [Fu2004] Dan Fu, "The Ultimate Guide to FSMs in Games", in AI Game Programming Wisdom 2, Charles River Media, 2004.
- [Fuchs] Carsten Fuchs, "Ca3D Engine", Carsten Fuchs Software, (<http://www.ca3d-engine.de>).
- [Funge1999] John David Funge, "AI for Games and Animation: a Cognitive Modeling Approach", A K Peters 1999.
- [Funge2004] John David Funge, "Artificial Intelligence for Computer Games", A.K. Peters, 2004.
- [Funkhouser1993] Thomas A. Funkhouser, Carlo H. Séquin, "Adaptive Display algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environment", University of California at Berkeley, 1993.
- [G.N.A.G.] G.N.A.G., "Live Orchestra Presentation", G.N.A.G.
- [Gambill2003] Tom Gambill, "Writing A Fast, Efficient, Fixed-Size Object Allocator", in Massively Multiplayer Game Development, Charles River Media, 2003.
- [Gamma1995] Erich Gamma, Ruchard Helm, Ralf Johnson, John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley Professional, 1995.
- [GarageGames-TNL] GarageGames-TNL, "Torque Network Library", GarageGames, (<http://www.opentnl.org/>).
- [GarageGames-Torque] GarageGames-Torque, "Torque Game Engine", GarageGames, (www.garagegames.com).

- [Garces-FOCA2006] Sergio Garces-FOCA, "Flexible Object-Composition Architecture", in Game Programming Wisdom 3, Charles River Media, 2006.
- [Garcés2006] Diego Garcés, "Scripting Language Survey", in Game Programming Gems 6, Charles River Media, 2006.
- [Garces2006] Sergio Garces, "Strategies for Multi-Processor AI", in AI Game Programming Wisdom 3, Charles River Media, 2006.
- [Gatlin2005a] Kang Su Gatlin, Pete Isensee, "OpenMP and C++: Reap the Benefits of Multithreading without All the Work", MSDN, (<http://msdn.microsoft.com/msdnmag/issues/05/10/OpenMP/>), 2005a.
- [Gatlin2005b] Kang Su Gatlin, "Optimization Best Practices", Meltdown, 2005b.
- [Gebhardt] Nikolaus Gebhardt, "Irrlicht Engine", (<http://irrlicht.sourceforge.net/>).
- [Gee2005] Kev Gee, "Optimizing Windows Games", Meltdown, 2005.
- [Geiger2000] C. Geiger, V. Paelke, W. Reinmann, W. Rosenbach, "Structured Design of Interactive Virtual and Augmented Reality Content", VE-Workshop, 2000.
- [Gerber2003] Richard Gerber, "Advanced OpenMP Programming", Intel Corp., 2003.
- [Germans2001] Desmond Germans, Hans Spoelder, Luv Reanmbot, Henri Bal, "VIRPI: A High-Level Toolkit for Interactive Scientific Visualization in Virtual Reality", IPT/EGVE, 2001.
- [Gilgenbach2006] Matt Gilgenbach, Travis McIntosh, "A Flexible AI System through Behavior Compositing", in AI Game Programming Wisdom 3, Charles River Media, 2006.
- [Gill2001] Nasib S. Gill, "Reusability Issues in Component-Based Development", 2001.
- [Gill2004] Sunbir Gill, "Visual Finite State Machine AI System", in Game Developer Magazine, CMP, November, 2004.
- [Glardon-CASA2004] Pascal Glardon-CASA, Ronan Boulic, Daniel Thalmann, "A coherent locomotion engine extrapolating beyond experimental data", In Proc. of Computer Animation and Social Agent, 2004.
- [Glardon2004] Pascal Glardon, Daniel Thalmann, "Pca-based walking engine using motion capture data", In Proc. of Computer Graphics International, 2004.
- [Glassenberg2006] Sam Z. Glassenberg, "Intro to Direct3D 10... and why the GPU will never be the same", GDC, 2006.
- [Glinker2004] Paul Glinker, "Fight Memory Fragmentation with Templated Freelists", Programming Gems 4, Charles River Media, 2004.
- [GLSL] GLSL, "The Shader Programming Language for OpenGL", (<http://oss.sgi.com/projects/ogl-sample/registry/ARB/GLSLangSpec.Full.1.10.59.pdf>).
- [Gobbetti1998] Enrico Gobbetti, Riccardo Scateni, "Virtual reality: Past, Present, and Future", Virtual Environments in Clinical Psychology and Neuroscience: Methods and Techniques in Advanced Patient-Therapist Interaction, pp. 3-20, 1998.
- [Gold2004] Julian Gold, "Object-Oriented Game Development", Addison-Wesley, 2004.
- [Gomez2001] Miguel Gomez, "Compressed Axis-Aligned Bounding Box Trees", in Game Programming Gems 2, Charles Rivers Media, 2001.
- [Goodmann1998] James R. Goodmann, "Using cache memory to reduce processor-memory traffic", 25 years of the international symposia on Computer architecture ACM Press, 1998.
- [Goodwin2005] Steve Goodwin, "Cross-Platform Game Programming", Charles River Media, 2005.
- [Gosselin2005] David R. Gosselin, Pedro V. Sanders, Jason L. Mitchell, "Drawing a Crowd", in ShaderX3: Advanced Rendering with DirectX and OpenGL, Charles River Media, 2005.
- [Goswami2001] Dhrubajyoti Goswami, Ajit Singh, Bruno Richard-Preiss, "Building Parallel Applications using Design Patterns", Advances in Software Engineering: Topics in Comprehension, Evolution and Evaluation, Springer-Verlag, pp. 243-265, 2001.
- [Gould2002] David Gould, "Complete Maya Programming", Morgan Kaufmann, 2002.
- [Grama2003] Ananth Grama, Anshul Gupta, Georgios Karypis, Vipin Kuman, "Introduction to Parallel Computing", Addison-Wesley, 2003.
- [Green-ART2005] Rod Green-ART, "Art Pipeline Philosophies for the Next Generation: A Technical Art Director's Perspective", in Game Developer Magazine, CMP, December, 2005.
- [Green2003] Robin Green, "Faster Math Functions", GDC, 2003.
- [Green2005] Susie Green, "Everything you Ever Wanted to Know about Normal Maps", GDCE, 2005.
- [Grimshaw1989] A. S. Grimshaw, "Real-Time Mentat: A Data-Driven Object-Oriented System", Proc IEEE Globecom, pp.232-241., 1989.
- [Gustafson1988] J.L. Gustafson, "Reevaluating Amdahl's Law", CACM, 31, pp. 532-533., 1988.
- [Gutek2003] Gerald L. Gutek, "Philosophical and Ideological Voices in Education", Allyn & Bacon, 2003.
- [Gutschmidt2003] Tom Gutschmidt, "Game Programming with Python, Lua, and Ruby", Premier Press, 2003.
- [Haigh-Hutchinson2005] Mark Haigh-Hutchinson, "Fundamentals of Real-Time Camera Design", GDC, 2005.
- [Haller2002] M. Haller, W. Hartmann, J. Zauner, "A Generic Framework for Game Development", SIGGRAPH, 2002.
- [Hamm2004] David Hamm, "Groovy Gravy: Tricking Out Your Custom Game Debugger", in Game Developer Magazine, CMP, September, 2004.
- [Handy1998] Jim Handy, "The Cache Memory Book", Morgan Kaufmann, 1998.
- [HANIM] HANIM, "HANIM Specification", (<http://www.h-Anim.org>).
- [Hannibal2000] Søren Hannibal, "3D Engines for Games: A Broader Perspective", Gamasutra, (http://www.gamasutra.com/features/20001013/hannibal_01.htm), 2000.
- [Harmon2005] Matthew Harmon, "Building Lua into Games", in Programming Gems 5, Charles River Media, 2005.
- [Harris2006] Mark Harris, "GPGPU Lessons Learned", 2006.
- [Havok] Havok, "Havok Complete", Havok, (<http://www.havok.com/>).
- [Hecker2000] Chris Hecker, Zachary Booth Simpson, "Game Programming Patterns & Idioms", Game Developer Magazine, CMP, September, 2000.
- [Hein2005] Marko Hein, "Gaming Power and Horse Power: A Faithful Partnership?" GDCE, 2005.

- [Hennessy2003] J.L. Hennessy, D.A. Patterson, "Computer Architecture: A Quantitative Approach", Morgan Kaufmann Publishers Inc., 2003.
- [Henney2003] Kevlin Henney, "C++ Threading: A Generic-Programming Approach", ACCU Spring Conference, 2003.
- [Hill2002] Mark Hill, Mikko Lipasti, "Cache Performance", University of Wisconsin-Madison, 2002.
- [Hill1992] Steve Hill, Academic Press, "A Simple Fast Memory Allocator", Graphics Gems III, Graphics Gems III, pp. 49-50, (<http://www.graphicsgems.org/>), 1992.
- [Hjelstrom2002] Greg Hjelstrom, Byron Garabrant, "Real-Time Hierarchical Profiling", Game Programming Gems 3, Charles River Media, 2002.
- [HLSL] HLSL, "The Microsoft DirectX High-Level Shader Programming Language", (<http://www.microsoft.com/directx/>).
- [Hoare1974] C.A.R. Hoare, "Monitors: An Operating System Structure Concept: The origin of Concurrent Programming from Semaphore to Remote Procedure calls", Springer, 1974.
- [Hodgson2004] David Hodgson, "Half-Life 2: Raising the Bar", Prima Games, 2004.
- [Hoefler2005] Torsten Hoefler, "The Cell Processor", 22nd Chaos Communication Congress, 2005.
- [Hoffert1998] Joe Hoffert, Kenneth Goldman, "Microthread: An Object for Behavioral Pattern for Managing Object Execution", PLoP, Distributed Programming Environments Group, 1998.
- [Hofstee2005] Peter H. Hofstee, "Introduction to the Cell Broadband Engine", IBM, 2005.
- [Huddy2004] Richard Huddy, "Helping Game Developers Make Great Games & 3Dc", European Developer Relation, ATI, 2004.
- [Hunt1999] Andrew Hunt, David Thomas, "The Pragmatic Programmer: From Journeyman to Master", Addison-Wesley Professional, 1999.
- [Hwang1993] Kai Hwang, "Advance Computer Architecture. Parallelism, Scalability, Programmability", McGraw Hill, 1993.
- [i-Glasses] i-Glasses, "i-Glasses HMD", (<http://www.i-glassesstore.com/>).
- [IA-DEPOT] IA-DEPOT, IA-DEPOT, (<http://www.ia-depot.com>).
- [IBM] IBM, "PowerPC Processors", (www.chips.ibm.com).
- [ICL] ICL, "Intel C++ Compiler", Intel Corporation, (<http://www.intel.com/cd/software/products/asmo-na/eng/compilers/index.htm>).
- [id-Q3Radiant] id-Q3Radiant, "Q3Radiant", id Software, (<http://www.qeradiant.com>).
- [idSoftware] idSoftware, "Doom III Engine", id Software, (<http://www.idsoftware.com/>).
- [IDV] IDV, "Speedtree", Interactive Data Visualization, (<http://www.speedtree.com/>).
- [Ierusalimschy1996] Roberto Ierusalimschy, "Lua-an extensible Extension Language", In Software: Practice & Experience, John Wiley & Sons Vol 26, 1996.
- [Ierusalimschy2003] Roberto Ierusalimschy, "Programming in Lua", Ingram, 2003.
- [IGDA-AI] IGDA-AI, "Special Interest Group on Artificial Intelligence", IGDA, (<http://www.igda.org/ai/>).
- [IGDA-MMO2004] IGDA-MMO, "Persistent Worlds Whitepaper", IGDA Online Game SIG, (http://www.igda.org/online/IGDA_PSW_Whitepaper_2004.pdf), 2004.
- [Infopath] Infopath, "Office InfoPath", (<http://office.microsoft.com/fr-fr/FX010857921036.aspx>).
- [Intel-VTune] Intel-VTune, "Intel VTune Performance Analyser", (<http://www.intel.com/software/products/vtune/>).
- [Intel] Intel, "Intel Corporation", (<http://www.intel.com>).
- [Intel2005] Intel, "Intel Multi-Core Processor Architecture Development Backgrounder", 2005.
- [Intel_ICL2005] Intel_ICL, "Intel C++ Compiler Optimizing Applications", Intel Corp., 2005.
- [Intel_Math] Intel_Math, "Intel Math Kernel Library", Intel, (<http://www.intel.com/cd/software/products/asmo-na/eng/perflib/mkl/index.htm>).
- [Intel_MT2003] Intel_MT, "Developing Multithreaded Applications: A Platform Consistent Approach", Intel, 2003.
- [Iprof] Iprof, "Iprof", (<http://silverspaceship.com/src/iprof/>).
- [Irish2005] Dan Irish, "The Game Producer Handbook: ", Premier Press, 2005.
- [Isakovic] Karsten Isakovic, "3D Engines List", (<http://cg.cs.tu-berlin.de/~ki/engines.html>).
- [Isensee-GDC2006] Pete Isensee-GDC, "C++ on Next-Gen Consoles: Effective Code for New Architectures", GDC, 2006.
- [Isensee2001] Pete Isensee, "C++ Optimization Strategies and Techniques", (<http://www.tantaloon.com/pete/cppopt/main.htm>), 2001.
- [Isensee2004] Pete Isensee, "Common C++ Performance Mistakes in Games", GDC, 2004.
- [Isensee2005] Pete Isensee, "Effective Use of OpenMP in Games", GDC, 2005.
- [Isensee2006] Pete Isensee, "Utilizing Multicore Processors with OpenMP", in Game Programming Gems 6, Charles River Media, 2006.
- [Itkonen2003] Juha Itkonen, "Measuring Object-Oriented Software Reusability", 2003.
- [Jacobs2005] Scott Jacobs, "Visual Design of State Machine", Game Programming Gems 5, Charles River Media, pp. 169-176, 2005.
- [Jacobson1999] Ivar Jacobson, Grady Booch, James Rumbaugh, "The Unified Software Development Process", Addison Wesley Professional, 1999.
- [Jade] Jade, "Java Adaptive Development Environment", (<http://jade.tilab.com/>).
- [Jaffe2005] Elliot Jaffe, "Scripting Languages", Hebrew University, 2005.
- [Jamieson2005] Alan C. Jamieson, Nicholas A. Kraft, Jason O. Hallstrom, Brian A. Malloy, "A Metric Evaluation of Game Application Software", The Future Play Conference, 2005.
- [Johnson2006] Geraint Johnson, "Goal Trees", in AI Game Programming Wisdom 3, Charles River Media, 2006.
- [Johnson1988] Ralph Johnson, Brian Foote, "Designing Reusable Classes", Journal of Object-Oriented Programming, SIGS, vol. 1, pp. 22-35, 1988.
- [Jones2003] Larry Jones, John Bergey, Matt Fisher, "Reducing System Acquisition Risk with Software Architecture Analysis and Evaluation", Ground System Architecture Workshop, 2003.
- [Jones1999] Michael T. Jones, "Scene Graph APIs: Wired or Tired?" Siggraph, Intrinsic Graphics, 1999.

- [Jones2005] Tim M. Jones, "AI Application Programming, 2nd Edition", Charles River Media, 2005.
- [JTC] JTC, "JThreads/C++", Orbacus, (www.orbacus.com/).
- [JUST] JUST, "JUST in time emergency interventions", EU Project, (<http://www.justweb.org/>).
- [Kaley1999] Danny Kaley, "Ansi/Iso C++ Professional Programmer's Handbook", Que, 1999.
- [Kallmann2001] Marcelo Kallmann, "Object Interaction in Real-Time Virtual Environments", Swiss Federal Institute of Technology - EPFL, DSc Thesis 2347, 2001.
- [Kallmann2003] Marcelo Kallmann, Bieri Hanspeter, Daniel Thalmann, "Fully Dynamic Constrained Delaunay Triangulations", Geometric Modelling for Scientific Visualization, 2003.
- [Kalogirou2005] Harry Kalogirou, "Multithreaded Game Scripting with Stackless Python", Thoughts Serializer, (<http://harkal.sylphis3d.com/2005/08/10/multithreaded-game-scripting-with-stackless-python/>), 2005.
- [Kane2004] Brad Kane, "Entertainment Experience First, Videogame Second: The Making of the Return of the King", GDC, (http://www.gamasutra.com/gdc2004/features/20040324/postcard-kane_03.shtml), 2004.
- [Kang2005] Kyoung-Don Kang, "Threads, SMP, and Microkernels", Watson School of Engineering and Applied Sciences, State University of New York at Binghamton, 2005.
- [Kantrowitz1997] M. Kantrowitz, "Fuzzy Logic and Fuzzy Expert System", 1997.
- [Karadimitriou2001] Kosmas Karadimitriou, "Coding Standards and Guidelines for Good Software Engineering Practice in C++", Louisiana State University, 2001.
- [Karlsson2005] Björn Karlsson, "Beyond the C++ Standard Library : An Introduction to Boost", Addison-Wesley Professional, 2005.
- [Kaspersky2003] Kris Kaspersky, "Code Optimization: Effective Memory Usage", A-List Publishing, 2003.
- [Katchabaw2004] Michael James Katchabaw, "The Future of Video Games", in CS437/CS641 Course Notes, Department of Computer Science, The University of Western Ontario, London, Ontario, Canada, 2004.
- [Kay1993] Alan C. Kay, "The Early History of Smalltalk", ACM SIGPLAN, 28, 1993.
- [Keith2006] Clinton Keith, "Agile Methodology in Game Development: Year 3", GDC, 2006.
- [Keith] Clinton Keith, "Agile Game Development", (<http://www.agilegamedevelopment.com/>).
- [Keller2006] Brian Keller, "XNA Studio: Introduction to XNA", GDC, 2006.
- [Kelso2001] John Kelso, Lance E. Arseneault, "DIVERSE: A Framework for Building Extensible and Reconfigurable Device Independent Virtual Environments", 2001.
- [Kessler2000] G.D. Kessler, D.A. Bowman, L.F. Hodges, "The Simple Virtual Environment Library: An Extensible Framework for Building VE Applications", MIT Press, 2000.
- [Khoo2002] Aaron Khoo, Robert. Zubek, "Applying Inexpensive AI Techniques to Computer Games", IEEE Intelligent Systems, 2002.
- [Khronos] Khronos, "OpenGL ES", Khronos, (<http://www.khronos.org/>).
- [Kiel-Perf2006] Jeff Kiel-Perf, "Performance Tools", GDC, 2006.
- [Kiel2006] Jeff Kiel, "OpenGL Performance Tools", GDC, 2006.
- [Kiessler] Oliver Kiessler, "Karma", (<https://karma.dev.java.net/>).
- [Kim2006] Yongha Kim, "Generating Globally Unique Identifiers for Game Objects", in Game Programming Gems 6, Charles River Media, 2006.
- [King2001] Yossarian King, "Floating-Point Tricks: Improving Performance with IEEE Floating Point", in Game Programming Gems 2, Charles River Media, 2001.
- [Kirby] John Brian Kirby, "SimBionic", Stottler Henke Associates, Inc., (<http://www.simbionic.com/>).
- [Klimovitski2001] Alex Klimovitski, Dean Macri, "SSE/SSE2 Toolbox: Solutions for Real-Life SIMD Problems", Meltdown, Intel Corporation, 2001.
- [Koch2000] Richard Koch, "The 80/20 Principle: The Secret of Achieving More With Less", Nicholas Breealey Publishing Ltd, 2000.
- [Koenig2000] Andrew Koenig, Barbara E. Moo, "Accelerated C++: Practical Programming by Example", Addison-Wesley Professional, 2000.
- [Koenig2006] David L. Koenig, "Faster File Loading with Access-Based File Reordering", In Game Programming Gems 6, Charles River Media, 2006.
- [Koeppel2002] F. Koeppel, "Massive Attack", Popular Science, 2002.
- [Krewell2004] Kevin Krewell, "High-Performance Processors: Trends and Implementations", Stat/MDR- Fall Processor Forum, 2004.
- [Krueger2006] Brian Krueger, Owen Brand, David Burton, "Reinventing Your Company Without Reinventing the Wheel", 2006.
- [Kruszewski2005] Paul A. Kruszewski, "A practical system for real-time crowd simulation on current and next-generation gaming platforms", 2005.
- [Kruszewski2006] Paul A. Kruszewski, "Real-Time Crowd Simulation Using AI.implant", in Game Programming Widsom 3, Charles River Media, 2006.
- [Kruszewski2004] Paul Kruszewski, "The Challenges and Follies of Building a Generic AI engine", AAAI, 2004.
- [Kushner2003] David Kushner, "It's a Mod, Mod World", IEEE Spectrum Careers, (<http://www.spectrum.ieee.org/careers/careerstemplate.jsp?ArticleId=i020203>), 2003.
- [Laird2000] John E. Laird, Michael van Lent, "Human-level AI's Killer Application: Interactive Computer Games", National Conference on Artificial Intelligence (AAAI) 2000.
- [Laird2001] John E. Laird, "Toward Human-level AI for Computer Games", AI Magazine, 2001.
- [Lakos1996] John Lakos, "Large-Scale C++ Software Design", Addison-Wesley, 1996.
- [LaMothe2005] Andre LaMothe, "The Black Art of Video Game Console Design", Sams, 2005.
- [Lange2003] Christian Lange, "Managing Game State Using A Database", in Massively Multiplayer Game Development, Charles River Media, 2003.

- [Lee-GDM2003] Jay Lee-GDM, "Data-Driven Subsystems for MMP Designers", in Game Developer Magazine, CMP, August, 2003.
- [Lee-OMP2003] Seyong Lee-OMP, "OpenMP: A Standard for Shared Memory Parallel Programming", 2003.
- [Lee2006] Matt Lee, Kev Gee, "Cross Platform Development Best Practices", GDC, 2006.
- [Lemarié2002] Lionel Lemarié, "Performance Analyser: Play your game cycle by cycle", SCEE Technology Group, 2002.
- [Lengyel2003] Eric Lengyel, "Mathematics for 3D Game Programming & Computer Graphics, Second Edition", Charles River Media, 2003.
- [Leuf2001] Bo Leuf, Ward Cunningham, "Wiki Way", Addison-Wesley, 2001.
- [Lewis2002] Michael Lewis, Jeffrey Jacobson, "Game Engines in Scientific Research ", Special issue on Game Engines in Scientific Research, Communication of ACM, 45, 2002.
- [Lindholm2001] E. Lindholm, Mark J. Kilgard, H. Moreton, "A User Programmable Vertex Engine", In Proceedings of SIGGRAPH 2001, ACM Press / ACM SIGGRAPH pp. 149-158, 2001.
- [Lippman1996] Stanley B. Lippman, "Inside the C++ Object Model", Addison Wesley Professional, 1996.
- [Lippman2005] Stanley B. Lippman, Josée Lajoie, Barbara E. Moo, "C++ Primer 4th Edition", Addison-Wesley Professional, 2005.
- [Llopis-C++2003] Noel Llopis-C++, "C++ For Game Programmers", Charles River Media, 2003.
- [Llopis-GDC2006] Noel Llopis-GDC, "Backwards Is Forward: Making Better Games with Test-Driven Development", GDC, 2006.
- [Llopis-GW] Noel Llopis-GW, "Games from Within: An engineering look at the game development process", (<http://www.gamesfromwithin.com/>).
- [Llopis-TDD2004] Noel Llopis-TDD, "Exploring the C++ Unit Testing Framework Jungle", (<http://www.gamesfromwithin.com/articles/0412/000061.html>), 2004.
- [Llopis-Web2004] Noel Llopis-Web, "Games from Within: Software Engineering", (http://www.gamesfromwithin.com/articles/cat_software_engineering.html), 2004.
- [Llopis2002] Noel Llopis, Brian Sharp, "Books, Solid Software Engineering for Games", (<http://www.convexhull.com/sweng/GDC2002.html>), 2002.
- [Llopis2003] Noel Llopis, "By the Books: Solid Software Engineering for Games", (<http://www.convexhull.com/sweng/GDC2003.html>), 2003.
- [Llopis2004] Noel Llopis, "The Beauty of Weak References and Null Objects", in Programming Gems 4, Charles River Media, 2004.
- [Llopis2006] Noel Llopis, Charles Nicholson, "Stay in the Game: Asset Hotloading for Fast Iteration", in Game Programming Gems 6, Charles River Media, 2006.
- [Lomont2006] Chris Lomont, "Floating-Point Tricks", in Game Programming Gems 6, Charles River Media, 2006.
- [Lua] Lua, "The Lua Programming Language".
- [Luabind] Luabind, "Luabind", (<http://luabind.sourceforge.net/>).
- [Lucasarts2003] Lucasarts, "Jedi Knight: Jedi Academy", Lucasarts, 2003.
- [Luebke2003] David Luebke, Martin Reddy, Jonathan D. Cohen, Amitabh and Watson Varshney, Benjamin and Huebner, Robert, "Level of Detail for 3D Graphics", Morgan Kaufmann Publishers, 2003.
- [MacKenzie1997] D. MacKenzie, R. Arkin, J. Cameron, "Multiagent mission specification and execution", Multiagent mission specification and execution, Autonomous Robots, pp. 29--52, 1997.
- [MacLellan2000] Jon MacLellan, "Half-Life Mods ", Gamespy, (<http://archive.gamespy.com/articles/november00/hlmods/>), 2000.
- [MacMahon2006] Mel MacMahon, "XNA Build: Advanced Concepts", GDC, 2006.
- [Madden2006] Blake Madden, "Using CppUnit To Implement Unit Testing", in Game Programming Gems 6, Charles River Media, 2006.
- [Magnemat-Thalmann] Nadia Magnemat-Thalmann, "Miralab", (<http://www2.miralab.unige.ch/>).
- [Magnemat-Thalmann2004] Nadia Magnemat-Thalmann, Daniel Thalmann, "Computer Animation", in Handbook of Computer Science, CRC Press, pp.40-1 – 40-19, 2004.
- [Malafeew2005] Eric Malafeew, "Data-Driven Programming Made Easy", GDC, (Malafeew), 2005.
- [Malakoff2004] Kevin Malakoff, "Gone is the game engine", Develop Magazine, Intent Media, January, 2004.
- [Malenfant2000] Didier Malenfant, "Writing Portable Code", GDC, 2000.
- [Mandel2004] Michael J. Mandel, "Vesatile and Interactive Virtual Humans: Hybrid use of Data-Driven and Dynamics-Based Motion Synthesis", Schoold of Computer Science, Carnegie Mellon University, 2004.
- [Mantis_Dev_Team] Mantis_Dev_Team, "Mantis: php/MySQL/web based bugtracking system", (<http://www.mantisbt.org/>).
- [Manzur] Ariel Manzur, "toLua++", (<http://www.codenix.com/~tolua/>).
- [Mao1998] Y. Mao, H.A. Sahraui, H. Lounis, "Reusability Hypothesis Verification Using Machine Learning Techniques: A Case Study", In Proceedings of the 13th IEEE International, 1998.
- [Maquire1999] Steve Maquire, "Debugging the Development Process : Practical Strategies for Staying Focused, Hitting Ship Dates, and Building Solid Teams", Microsoft Press, 1999.
- [Marselas2000] Herb Marselas, "Profiling, Data Analysis, Scalability, and Magic Numbers, Part 2: Using Scalable Features and Conquering the Seven Deadly Performance Sins", Gamasutra, (http://www.gamasutra.com/features/20000816/marselas_01.htm), 2000.
- [Marselas2003] Herb Marselas, "Age of Mythology Graphics Compatibility", Meltdown, 2003.
- [Mattson2004] Timothy Mattson, Beverly Sanders, Berna Massingill, "Patterns for Parallel Programming", Addison Wesley Professional, 2004.
- [Maughan2003] Chris Maughan, "All The Polygons You Can Eat", Nvidia, 2003.

- [Maughan2006] Christopher Maughan, Kevin Bjorke, "Cross-Platform Shader Development with FX Composer 2", GDC, 2006.
- [McConnel1996] Steve McConnel, "Rapid Development", Microsoft Press, 1996.
- [McConnel2006] Steve McConnel, "Software Estimation: Demystifying the Black Art", Microsoft Press, 2006.
- [McConnell2004] Steve McConnell, "Code Complete, Second Edition", Microsoft Press, (<http://www.cc2e.com/>), 2004.
- [McKay2000] Everett N. McKay, Mike Woodring, "Debugging Windows Programs: Strategies, Tools, and Techniques for Visual C++ Programmers", Addison-Wesley Professional, 2000.
- [McLean2002] Alex W. McLean, "An Efficient AI Architecture Using Prioritized Task Categories", AI Game Programming Wisdom, Charles River Media, 2002.
- [McNaughton2006] Matthew McNaughton, Thomas Roy, "Creating a Visual Scripting System", in AI Game Programming Wisdom 3, Charles River Media, 2006.
- [McShaffry2005] Mike McShaffry, "Game Coding Complete, 2nd Edition", Paraglyph Press, 2005.
- [Merceron2004] Julien Merceron, "The Impact of Middleware Technologies on your Game Development", GDC, 2004.
- [Metrowerk-Analysis] Metrowerk-Analysis, "CodeWarrior Analysis Tools", Metrowerk, (<http://www.metrowerks.com/>).
- [Meyers-EffecSTL2001] Scott Meyers-EffecSTL, "Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library", Addison Wesley Professional, 2001.
- [Meyers-Op1994] Scott Meyers-Op, "Writing Effective Versions of new and delete in C++", Object Expo, 1994.
- [Meyers-Perf1994] Scott Meyers-Perf, "High-Performance C++ Programming", OOPSLA, 1994.
- [Meyers-STL2001] Scott Meyers-STL, "Maximizing STL Performance", The C++ Seminar I, 2001.
- [Meyers1994] Scott Meyers, "Writing Efficient C++ Programs", Software Development, 1994.
- [Meyers1995] Scott Meyers, "Using Inheritance Effectively in C++", Windows Solutions, 1995.
- [Meyers1997] Scott Meyers, "Smart Pointers A to Z", Visual C++ Developers Conference, 1997.
- [Meyers2001] Scott Meyers, "STL: 50 Specific Ways to Improve Your Use of the Standard Template Library", Addison Wesley Professional, 2001.
- [Meyers2002] Scott Meyers, "STL Allocators", Software Development, 2002.
- [Meyers2005] Scott Meyers, "Effective C++: 55 Specific Ways to Improve Your Programs and Designs, 3rd Edition", Addison Wesley Professional, 2005.
- [Meyers2006] Scott Meyers, "An Overview of TR1", Software Development, 2006.
- [Microsoft-SS] Microsoft-SS, "Visual SourceSafe", Microsoft, (<http://msdn.microsoft.com/vstudio/previous/ssafe/>).
- [Microsoft-XNA] Microsoft-XNA, "XNA", Microsoft, (<http://www.microsoft.com/xna/>).
- [Microsoft2005] Microsoft, "Methods and system for general skinning via hardware accelerators ", United States Patent Application, 2005.
- [Miller2005] Russ Miller, Laurence Boxer, "Algorithms Sequential and Parallel: A Unified Approach, Second Edition ", Charles River Media, 2005.
- [Milo2005] Yip Milo, "Middleware in Game Development ", Europe-China Workshop on E-Learning and Games (Edutainment), 2005.
- [Moltenbrey2004] K. Moltenbrey, "Digital artists re-create ancient Rome for the epic miniserie Spartacus", Computer Graphics World, 2004.
- [Moore1965] Gordon E. Moore, "Cramming more components onto integrated circuits", Electronic Magazine 38, pp. 114-117, 1965.
- [Moore2002] Gordon E. Moore, "No Exponential is Forever", Intel WSTS, 2002.
- [MORGAN] MORGAN, "MORGAN: A distributed multi-user framework for VR/AR applications", Fraunhofer FIT, (http://www.fit.fraunhofer.de/gebiete/mixed-reality/index_en.xml?aspect=morgan).
- [Morris2006] Chris Morris, "Gaming's biggest least-known company: Players and developers know Epic Games. It's time you did as well." Game Over Weekly Column, CNN, (http://money.cnn.com/2006/03/24/commentary/game_over/gdc_epic/), 2006.
- [MS-GS] MS-GS, "Microsoft Game Studios", Microsoft, (<http://www.microsoft.com/games/default.aspx>).
- [MS_CL] MS_CL, "Microsoft Visual C++ Compiler", Microsoft Corporation, (<http://msdn.microsoft.com/visualc/>).
- [MS_Team_System2005] MS_Team_System, "Microsoft Visual Studio 2005 Team System", Microsoft, (<http://msdn.microsoft.com/vstudio/teamsystem/default.aspx>), 2005.
- [MSDN-HT2004] MSDN-HT, "Windows Support for Hyper-Threading Technology ", (<http://www.microsoft.com/whdc/system/CEC/HT-Windows.mspx>), 2004.
- [MSDN_LT2002] MSDN_LT, "Less Install, More Game: Reducing Game Install and Load Times, Without Reducing Content", GDC, 2002.
- [Murdock2005] Kelly Murdock, "3ds max 7 Bible", Wiley, 2005.
- [Myers2006] Kevin Myers, "D3D10: Prepping your Engine for a smooth start", GDC, 2006.
- [Nettle2000] Paul Nettle, "Generic Collision Detection for Games Using Ellipsoids", 2000.
- [Nevrax] Nevrax, "Nel", Nevrax, (<http://www.nevrax.org/tikiwiki/tiki-index.php>).
- [NewTek] NewTek, "LightWave", NewTek, (<http://www.newtek.com/lightwave/>).
- [Newton] Newton, "Newton Game Dynamics SDK", Newton Game Dynamics, (<http://physicsengine.com/>).
- [Niemeyer] Gustavo Niemeyer, "LunaticPython", (<https://moin.conectiva.com.br/LunaticPython>).
- [Nintendo-DS2004] Nintendo-DS, <http://www.nintendo.com/systemsds>, "Nintendo DS", Nintendo, 2004.
- [Nintendo-Revolution2006] Nintendo-Revolution, "Revolution", Nintendo, (<http://www.nintendo.com/revolution>), 2006.
- [Nischt2006] Michael Nischt, Elisabeth André, "Real-Time Character Animation on the GPU", in ShaderX4: Advanced Rendering Techniques, Charles River Media, 2006.
- [Novales2006] Erik Novales, "The Neverwinter Nights 2 Toolset: A Case Study of Tools Development", GDC, 2006.
- [NVIDIA-NVPerfKit] NVIDIA-NVPerfKit, "NVPerfKit", nVIDIA, (http://developer.nvidia.com/object/nvperfkit_home.html).

- [NVIDIA-NVSG] NVIDIA-NVSG, "NVSG", Nvidia, (http://developer.nvidia.com/object/nvsg_home.html).
- [O'Donnel2001] Martin O'Donnel, "Producing audio for Halo", MS Press, 2001.
- [Oberhumer] Markus F.X.J. Oberhumer, "LZO: a portable lossless data compression library written in ANSI C." (<http://www.oberhumer.com/opensource/lzo/>).
- [Oliveira2002] M. Oliveira, "The Java Adaptive Dynamic Environment", UCL, (<http://www.cs.ucl.ac.uk/staff/M.Oliveira/research/jade.htm>), 2002.
- [Olsen2000] John Olsen, "Stats: Real-Time Statistics and In-Game Debugging", in Game Programming Gems, Charles River Media, 2000.
- [Olukotun2005] Kunle Olukotun, Lance Hammond, "The Future of Microprocessors", in ACM Queue, vol. 3, September, 2005.
- [OMG] OMG, "Object Management Group", (<http://www.omg.org>).
- [OpenGL] OpenGL, "OpenGL 2", (<http://www.opengl.org>).
- [OpenMP] OpenMP, "OpenMP", (<http://www.openmp.org>).
- [OpenThreads] OpenThreads, "OpenThreads: A cross-platforms, lightweight C++ thread API", (<http://openthreads.sourceforge.net/>).
- [Osfield] Robert Osfield, "OpenSceneGraph", (<http://www.openscenegraph.org/>).
- [osgConv] osgConv, "osgConv: utility program for reading in 3d databases and convert them into optimized files format", (<http://www.openscenegraph.org/osgwiki/pmwiki.php/UserGuides/Osgconv>).
- [osgEdit] osgEdit, "osgEdit: An open editor for an open scenegraph", (<http://osgedit.sourceforge.net/>).
- [Ousterhout1998] John K. Ousterhout, "Scripting: Higher Level Programming for the 21st Century", IEEE Computer, (<http://home.pacbell.net/ouster/scripting.html>), 1998.
- [Owens2005] John Owens, "Streaming Architecture and Technology Trends", in GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose, Addison Wesley, 2005.
- [Pallister1999] Kim Pallister, Dean Macri, "Building Scalable 3D Games for the PC", Gamasutra, (http://www.gamasutra.com/features/19991124/pallisternacri_01.htm), 1999.
- [Palmer2005] Grant Palmer, "Physics for Game Programmers", Apress, 2005.
- [Papagiannakis2004] George Papagiannakis, Sebastien Schertenleib, Michal Ponder, Nadia Magnenat-Thalmann, Daniel Thalmann, "Real-Time Virtual Humans in AR Sites", Proc. IEE Visual Media Production (CVMP), pp. 273-276, 2004.
- [Papagiannakis2005] George Papagiannakis, Sébastien Schertenleib, Brian O'Kennedy, Nadia Magnemat-Thalmann Nadia Arevalo-Poizat Marlene, Andrew Stoddart, Daniel Thalmann, "Mixing virtual and real scenes in the site of ancient Pompeii", Computer Animation and Virtual Worlds, vol. 16, pp.11-24., 2005.
- [PAPI2004] PAPI, "Performance application programming interface", <http://icl.cs.utk.edu/projects/papi>, 2004.
- [Parikh1983] Girish Parikh, Nicholas Zvegintzov, "Tutorial on Software Maintenance", IEEE Computer Society Press, 1983.
- [Parise2005] Jon Parise, "Massively Multiplayer Scripting Systems", in Massively Multiplayer Game Development 2, Charles River Media, 2005.
- [Parisi2003] Tony Parisi, "FLUX: Lightweight Web Graphics in XML", Siggraph, 2003.
- [Park1992] Robert E. Park, with the Size Subgroup of the Software Metrics Definition Working Group and the Software Process Measurement Project Team, "Software Size Measurement: A Framework for Counting Source Statements", CMU/SEI, 1992.
- [Parnas1985] D.L. Parnas, P.C. Clements, D.M. Weiss, "The Modular Structure of Complex Systems", IEEE Transaction Software Engineering, 259-266, 1985.
- [Patterson1997] D.A. Patterson, John L. Hennessy, "Computer Organization and Design Second Edition : The Hardware/Software Interface", Morgan Kaufmann, 1997.
- [Paul1995] R.A. Paul, "Metric-Guided Reuse", In proceedings of 7th International Conference on tools with artificial Intelligence (TAI'95), pp. 120-127, 1995.
- [Penton2003] Ron Penton, "Data Structures for Game Programmers", Premier Press, 2003.
- [Perforce-Jam] Perforce-Jam, "Jam Build System", Perforce, (<http://www.perforce.com/jam/jam.html>).
- [Perforce] Perforce, "Perforce SCM", Perforce Software, (<http://www.perforce.com/>).
- [Performance_Tools_Lab2005] Performance_Tools_Lab, "Intel VTune Performance Analyser", Siggraph, 2005.
- [Perl] Perl, "Perl Scripting Language", (<http://www.perl.com/>).
- [Perlin1985] Kevin Perlin, "An Image Synthesizer", Computer Graphics, 1985.
- [Perminov2004] Maxim Perminov, Aaron Coday, Will Damon, "Real World Multithreading in PC Games: Case Studies", GDC, 2004.
- [Pettre2005] Julien Pettre, J.-P. Laumond, Daniel Thalmann, "A navigation graph for real-time crowd animation on multilayered and uneven terrain", Proc. The First International Workshop on Crowd Simulation (V-CROWDS'05), 2005.
- [PGR32005] PGR3, "Project Gotham Racing 3", Bizarre Creations, Microsoft Game Studios, (<http://www.bizarreonline.net/>), 2005.
- [Planetside] Planetside, "Terragen", Planetside Software, (<http://www.planetside.co.uk/terragen/>).
- [Plumb2006] Jason Plumb, "Exploiting Scalable Multi-Core Performance in Large Systems", GDC, 2006.
- [Plummer2004] Jeff Plummer, "A Flexible and Expandable Architecture for Computer Games", Arizona State University, 2004.
- [Poiker2002] Falko Poiker, "Creating Scripting Languages for Nonprogrammers", in AI Game Programming Wisdom, Charles River Media, 2002.
- [Polya1957] G. Polya, "How to solve it." Doubleday and Co., Inc., 1957.
- [Polyphony2004] Polyphony, "The Making of GT4", Sony, 2004.

- [Ponder2004] Michal Ponder, "Component-Based Methodology and Development Framework for Virtual and Augmented Reality Systems", Phd Thesis, EPFL, 2004.
- [Pree1994] Wolfgang Pree, "Design Patterns for Object-Oriented Software Development", Addison-Wesley, 1994.
- [Price2004] Mark T. Price, "Using XML Without Sacrificing Speed", in Programming Gems 4, Charles River Media, 2004.
- [Python] Python, "Python Programming Language", (<http://www.python.org>).
- [Quantum3D] Quantum3D, "VTree", Quantum3D, (<http://www.quantum3d.com/products/Software/vtree.html>).
- [Qwt] Qwt, "Qwt: Qt Widgets for Technical Applications", (<http://qwt.sourceforge.net/index.html>).
- [Rabin-AI2004] Steve Rabin-AI, "Common Game AI Technique", in AI Game Programming Widsom 2, Charles River Media, 2004.
- [Rabin2000] Steve Rabin, "Real-Time In-Game Profiling", in Game Programming Gems, Charles River Media, 2000.
- [Rabin2002] Steve Rabin, "IA Game Programming Wisdom", Charles River Media, 2002.
- [Rabin2004] Steve Rabin, "The Science of Debugging Games", Programming Gems 4, Charles River Media, 2004.
- [Rabin2005] Steve Rabin, "Introduction to Game Development", Charles River Media, 2005.
- [RAD] RAD, "The Miles Sound System", RAD Game Tools, (<http://www.radgametools.com/miles.htm>).
- [Radiant] Radiant, "Radiant AI", Bethesda Software.
- [Radon] Radon, "The Nebula Device", Radon Labs GmbH, (<http://nebuladevice.cubik.org/>).
- [Rajwar2002] Ravi Rajwar, "Speculation-Based Techbique for Lock-Free Execution of Lock-Based Programs", University of Wisconsin, Madison, 2002.
- [Ramsey2005] Michael Ramsey, "Parallel AI Development with PVM", in Programming Gems 5, Charles River Media, 2005.
- [Ranck2000] Steven Ranck, "Frame-Based Memory Allocation", in Game Programming Gems, Charles River Media, 2000.
- [Ratcliff2005] John W. Ratcliff, "Optimization Techniques for Rendering Massive Quantities of Mesh Deformed Characters in Real-Time", in Massively Multiplayer Game Development 2, Charles River Media, 2005.
- [Rational] Rational, "Rational Rose", IBM, (<http://www-306.ibm.com/software/rational/>).
- [Read2003] Daniel Read, "Software Design and Programmers", developer.*, (http://www.developerdotstar.com/mag/articles/read_designprog.html), 2003.
- [Reimer2005] Jeremy Reimer, "Cross-platform game development and the next generation of consoles", Arstechnica, 2005.
- [Reiners2002] Dirk Reiners, "A Flexible and Extensible Traversal Framework for Scenegraph Systems", OpenSG, 2002.
- [RenderWare_AI] RenderWare_AI, "RenderWare A.I." Criterion Software, (<http://www.renderware.com/ai.asp>).
- [Rene2005] Bjarne Rene, "Component Based Object Management", Game Programming Gems 5, Charles River Media, pp. 25-37, 2005.
- [Reynolds-Steer1999] Craig W. Reynolds-Steer, "Steering Behaviors For Autonomous Characters", GDC, 1999.
- [Reynolds2006] Craig W. Reynolds, "Crowd Simulation on PS3", GDC, 2006.
- [Rigo] Armin Rigo, "Quark", (<http://sourceforge.net/projects/quark/>).
- [Riguer2002] Guennadi Riguer, "Performance Optimization technique for ATI Graphics Hardware with DirectX9", Ati Press, 2002.
- [Riguer2006] Guennadi Riguer, "How to Work on Next Gen Effects Now: Bridging DX10 and DX9", GDC, 2006.
- [Riley2003a] Scott Riley, "Game Programming With Python ", Charles River Media, 2003a.
- [Riley2003b] Sean Riley, "Data-Driven System For MMP Games", in Massively Multiplayer Game Development, Charles River Media, 2003b.
- [Robbins2003] Jonathan Robbins, "Simulation Level of Detail Or Doing As Little Work As Possible", Physically Based Modeling, Simulation, and Animation, 2003.
- [Rollings2000] Andrew Rollings, Morris Dave, "Game Architecture and Design", The Coriolis Group, 2000.
- [Rubin2003] Jason Rubin, "Video Games Graphics: Who care's?" GDC, 2003.
- [Rucker2002] Rudy Rucker, "Software Engineering and Computer Games", Essex, United Kingdom, Addison Wesley, 2002.
- [Sane1995] Aamod Sane, "The Elements of Pattern Style", University of Illinois at Urbana-Champaign, 1995.
- [SCEE2000] SCEE, "EE User's Manual", London, England, Sony Computer Entertainment Europe, 2000.
- [SCEE2002] SCEE, "Performance Analyser", SCEE, 2002.
- [Schach1993] S.R. Schach, "Software Engineering", Asken Associates, 1993.
- [Schemenaur2001] N. Schemenaur, T. Peter, M. Hetland, "Simple Generators", PEP, 2001.
- [Schertenleib2002] Sebastien Schertenleib, "Complex 3D Environments System Rendering and consistent frame rate", 2002.
- [Schertenleib2004] Sebastien Schertenleib, Mario Gutierrez, Frederic Vexo, Daniel Thalmann, "Conducting a Virtual Orchestra: Multimedia Interaction in a Music Performance Simulator", IEEE Multimedia, Special Issue on Multisensory Communication and Experience through Multimedia, 11, pages. 40-49, 2004.
- [Schimdt2000] Douglas C. Schimdt, Michael Stal, Hans Rohnert, Frank Buschmann, "Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects", 2000.
- [Schinners2005] Pete Schinners, "Pygame: set of Python modules designed for writing games and VR applications", (<http://www.pygame.org/>), 2005.
- [Schulte2002] Mike Schulte, "Lecture:15: Calculating and Improving Cache Performance", Computer Architecute Course, Lehigh University, 2002.
- [Schultz2005] Charles R. Schultz, Robert Bryant, Tim Langdell, "Game Testing All in One", Premier Press, 2005.
- [Schuytema2005] Paul Schuytema, Mark Manyen, "Game Development with Lua", Charles River Media, 2005.
- [Schwab2004] Brian Schwab, "AI Game Engine Programming", Charles River Media, 2004.
- [Scons] Scons, "Scons: A software contrustion tool", (<http://www.scons.org/>).
- [Scott2002] Bob Scott, "Architecture a Game AI", in AI Game Programming Widsom, Charles River Media, 2002.
- [Scrum] Scrum, "The Scrum Development Process", Mountain Goat Software, (<http://www.mountaingoatsoftware.com/scrum/>).

- [Sebek2002] Filip Sebek, "Instruction Cache Memory Issue in Real-Time Systems", Department Of Computer Science and Engineering, 2002.
- [Sedgewick1990] Robert Sedgewick, "Algorithms in C", Addison Wesley, 1990.
- [Sense8] Sense8, "WorldToolKit", Sense8, (<http://sense8.sierraweb.net/products/wtk.html>).
- [Serena] Serena, "Serena Version Manager", Serena, (<http://www.serena.com/Products/professional/vm/home.asp>).
- [Serious-Games] Serious-Games, "Serious Games Initiative", (<http://www.seriousgames.org/>).
- [SGI] SGI, "Performer", (<http://www.sgi.com/products/software/performer/>).
- [Shaw2004] Chris Shaw, "AI in video Games", CS 4455: Video Game Design and Programming, Georgia Tech, 2004.
- [Shewchuk1997] Jonathan Shewchuk, "Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates", Discrete & Computational Geometry, 18, pp. 305-363 (<http://www.cs.cmu.edu/~quake-papers/robust-arithmetic.ps>), 1997.
- [Shi1996] Yuan Shi, "Reevaluating Amdahl's Law and Gustafson's Law", Temple University, 1996.
- [Shumaker2004] Scott Shumaker, "Techniques and Strategies for Data-driven design in Game Development ", Computer and Information Science, 2004.
- [Simonyi1999] Charles Simonyi, "Hungarian Notation", MSDN, (<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs600/html/hunganotat.asp>), 1999.
- [Simpson2002] Jake Simpson, "Game Engine Anatomy", ExtremeTech, 2002.
- [Simpson1999] Zachary Booth Simpson, "Design Pattern for Computer Games", GDC, (<http://www.mine-control.com/zack/patterns/gamepatterns.html>), 1999.
- [Singh2003] Montek Singh, "Computer Architecture and Implementation", Comp 206, 2003.
- [SIP2005] SIP, "A tool that makes it very easy to create Python bindings for C and C++ libraries. " (<http://www.riverbankcomputing.co.uk/sip/index.php>), 2005.
- [Smith1987] A.J. Smith, "Line (block) size choice for CPU cache memories", IEEE Transactions on Computers, 1987.
- [Smith] Russell Smith, "Open Dynamics Engine", (<http://www.ode.org/>).
- [SN-Systems] SN-Systems, "Tuner", (<http://www.snsys.com/>).
- [Soar] Soar, "Soar: a general cognitive architecture for developing systems that exhibit intelligent behavior", (<http://sitemaker.umich.edu/soar>).
- [Soconne] Soconne, "FreeWorld3D", Soconne Software, (<http://www.freeworld3d.org/>).
- [Softimage-AI] Softimage-AI, "Softimage XSI Behavior: Intelligent Digital Choreography", Avid Technologies, (<http://www.softimage.com/products/behavior/v2/default.asp>).
- [Softimage] Softimage, "XSI", Softimage, (<http://www.softimage.com/home/>).
- [Sonnenschein2001] David Sonnenschein, "Sound Design", Studio City, Michael Weise Production, 2001.
- [Sqlite] Sqlite, "Sqlite: a small C library that implements a self-contained, embeddable, zero-configuration SQL database engine", (<http://www.sqlite.org/>).
- [St-Laurent2005] Sebastien St-Laurent, Wolfgang Engel, "The COMPLETE Effect and HLSL Guide", Paradoxal Press, 2005.
- [Stenson2006] Richard Stenson, "Collada for the PlayStation3", GDC, 2006.
- [Stevens1974] W.P. Stevens, G.J. Myers, L.L. Constantine, "Structured Design", IBM Systems Journal, 13, pp. 115-139, 1974.
- [Stottler2003] Henke Stottler, "History of AI", (http://www.stottlerhenke.com/ai_general/history.htm), 2003.
- [Stout] Bryan Stout, John O'Brien, "Embodied Agents in Computer Games", GDC.
- [Stoy2006] Chris Stoy, "Game Object Component System", in Game Programming Gems 6, Charles River Media, 2006.
- [Strauss1992] P. Strauss, R. Carey, "Inventor: An Object-Oriented 3D Graphics Toolkit", SIGGRAPH 1992.
- [Streetering] Steeve Streetering, "Ogre3D", (<http://www.ogre3d.org/>).
- [Stroustrup-Reuse1996] Bjarne Stroustrup-Reuse, "Language-technical Aspects of Reuse", Proc. 4th International Conference on Software reuse, pp. 11-19, 1996.
- [Stroustrup1991] Bjarne Stroustrup, "What is Object-Oriented Programming?" Proc. 1st European Software Festival, 1991.
- [Stroustrup1994] Bjarne Stroustrup, "The Design and Evolution of C++", Addison Wesley, 1994.
- [Stroustrup1996] Bjarne Stroustrup, "A perspective on Concurrency and C++", in Parallel Programming using C++, The MIT Press, 1996.
- [Stroustrup1997] Bjarne Stroustrup, "The C++ programming language", Addison Wesley, 1997.
- [Stroustrup2005] Bjarne Stroustrup, "The Design of C++0X: Reinforcing C++'s proven strenghts, while moving into the future", C/C++ User Journal, CMP, 2005.
- [Stroustrup2006] Bjarne Stroustrup, "The C++ Source: A Brief Look at C++0x", artima developer, (<http://www.artima.com/cppsource/cpp0x.html>), 2006.
- [Stutz2004] David Stutz, "The Natural History of Software Platforms ", (http://www.synthesist.net/writing/software_platforms.html), 2004.
- [Sudden-Presence2004] Sudden-Presence, "XDS: The eXtensible Data Stream", Sudden Presence, (<http://www.suddenpresence.com/xds/>), 2004.
- [Sugimoto] Tomohiko Sugimoto, "KASHMIR 3D ", (<http://www.kashmir3d.com/index-e.html>).
- [Sun-Java3D] Sun-Java3D, "Java3D", Sun, (<http://java.sun.com/products/java-media/3D/>).
- [Sun-JavaDoc] Sun-JavaDoc, "JavaDoc", Sun, (<http://java.sun.com/j2se/javadoc/writingdoccomments/>).
- [Sun] Sun, "UltraSPARC Processors", (<http://www.sun.com/processors/>).
- [Sung2004] Mankyu Sung, Michael Gleicher, Stephen Chenney, "Scalable behaviors for crowd simulation", Eurographics, vol. 23, 2004.
- [Sutherland1965] I. E. Sutherland, "The ultimate display", In Proceedings of IFIPS Congress (New York City, NY, May 1965), vol. 2, pp. 506-508, 1965.

- [Sutter-PDC2005] Herb Sutter-PDC, "C++: Future Directions in Language Innovation", Microsoft Professional Developers Conference (PDC), 2005.
- [Sutter2004] Herb Sutter, Andrei Alexandrescu, "C++ Coding Standards : 101 Rules, Guidelines, and Best Practices", Addison-Wesley Professional, 2004.
- [Sutter2005] Herb Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software", Dr. Dobbs's Journal, CMP, 2005.
- [SVN] SVN, "SVN: Subversion", (<http://subversion.tigris.org/>).
- [Swartz2004] Bill Swartz, "Follow The Money: Understanding Console Publishers", GDC, 2004.
- [Sweeney-GDC2006] Tim Sweeney-GDC, Martin Sweitzer, "Building a Flexible Game Engine: Abstraction, Indirection, and Orthogonality", 2006.
- [Sweeney1998] Tim Sweeney, "UnrealScript Language Reference", Epic Games, (<http://unreal.epicgames.com/UnrealScript.htm>), 1998.
- [Sweeney2005] Tim Sweeney, "Game Technology and Content Creation for the Next Generation", GDC, 2005.
- [Sweeney2006] Tim Sweeney, "The Next Mainstream Programming Language: A Game Developer's Perspective", in the 33rd Annual Symposium on Principles of Programming Languages (PoPL), ACM, 2006.
- [Sweetser2003] P. Sweetser, "Current AI in Games: A Review", Australian Journal of Intelligent Information Processing Systems, 2003.
- [SWIG] SWIG, "The SWIG Library", (<http://www.swig.org>).
- [Szécsi2003] Lazlo Szécsi, "An effective Implementation of the K-D Tree", in Graphics Programming Methods, Charles River Media, 2003.
- [Szyperski2002] Clemens Szyperski, "Component Software: Beyond Object-Oriented Programming, 2nd Edition", Addison Wesley Professional, 2002.
- [Tatarchuk2005] Natalya Tatarchuk, "Richer Worlds for Next Gen Games: Data Amplification Techniques Survey", GDCE, 2005.
- [Teixera de Sousa2002] Bruno Miguel Teixeira de Sousa, "Game Programming All in One", Premier Press, 2002.
- [Telles2001] Matthew A. Telles, Yuan Hsieh, "The Science of Debugging", Coriolis Group Books, 2001.
- [Thalmann-SIGG2004] Daniel Thalmann-SIGG, C. Hery, H. Ono, D. Sutton, S. Lippman, S. Regelous, "Crowd and Group and Crowd Animation", SIGGRAPH 2004.
- [Thalmann] Daniel Thalmann, "VRLab", (<http://vrlab.epfl.ch>).
- [Thalmann2004] Daniel Thalmann, Nadia Magnemat-Thalmann, S. Donikian, "Automatic Generation of Animated Population in Virtual Environments", Eurographics, 2004.
- [Thatcher2002] Ulrich Thatcher, "Chunked LOD", (<http://tulrich.com/geekstuff/chunklod.html>), 2002.
- [Thomason2005] Andy Thomason, "Writing Efficient Game Code for Next-Gen Console Architectures", Gamasutra, (http://www.gamasutra.com/features/2005/1220/thomason_01.shtml), 2005.
- [Tismer2000] C. Tismer, "Continuations and Stackless Python", In Proceedings of the 8th International Python Conference, Arlington, VA, 2000.
- [TOGAF] TOGAF, "The Open Group Architecture Framework", (<http://www.opengroup.org/architecture/>).
- [Tokamak] Tokamak, "Tokamak Game Physics SDK", Tokamak, (<http://www.tokamakphysics.com/>).
- [TouchdownEntertainment] TouchdownEntertainment, "JupiterEX", Touchdown Entertainment, (<http://www.touchdownentertainment.com/jupiterEX.htm>).
- [Tozour-AI2002] Paul Tozour-AI, "The Evolution of Game AI", in AI Game Programming Wisdom, Charles River Media, 2002.
- [Tozour2002] Paul Tozour, "The Perils of AI Scripting", in AI Game Programming Widsom, Charles River Media, 2002.
- [Trebilco2006] Damien Trebilco, "GLSL Shader Debugging with GLIntercept", in ShaderX4: Advanced Rendering Techniques, Charles River Media, 2006.
- [Tremblay2004] Christopher Tremblay, "Mathematics for Game Developers", Premier Press, 2004.
- [Trolltech] Trolltech, "QT", (<http://www.trolltech.com>).
- [Turner] Paul D. Turner, "Crazy Eddie's GUI System", (<http://www.cegui.org.uk>).
- [UCSC] UCSC, "BAMBOO: Automatic Generation of SCM Systems", UCSC Engineering, (<http://www.soe.ucsc.edu/research/labs/grase/bamboo/>).
- [Ulicny-Brush2004] Branislav Ulicny-Brush, Pablo de Heras Ciechowski, Daniel Thalmann, "Crowdbush: Interactive Authoring of Real-time Crowd Scenes", Proc. ACM SIGGRAPH/Eurographics Symposium on Computer Animation pp. 243-252, 2004.
- [Ulicny2005] Branislav Ulicny, "Crowd for Interactive Virtual Environments", Phd Thesis, EPFL, 2005.
- [UML] UML, "Unified Modeling Language", (<http://www.uml.org>).
- [Unibrain2004] Unibrain, "Unibrain firewire camera", (<http://www.unibrain.com/home/>), 2004.
- [UnitTest++] UnitTest++, "UnitTest++: a lightweight unit testing framework for C++." (<http://unittest-cpp.sourceforge.net/>).
- [Valdes2004] Robert Valdes, "In the Mind of the Enemy: The Artificial Intelligence of Halo 2", Stuffo, 2004.
- [Valve-Hammer] Valve-Hammer, "Hammer", Valve, (<http://collective.valve-erc.com/index.php?go=hammer>).
- [Valve-Source2004] Valve-Source, "Valve Source Engine", Valve Software, (<http://www.valvesoftware.com/sourcelicense/>), 2004.
- [van der Beek2003] Jelle van der Beek, "Dependency Graphs in Games", Gamasutra, 2003.
- [van Emde Boas1977] P. van Emde Boas, "Preserving order in a forest in less than logarithmic time and linear space", Inf. Process. Lett., pp. 80-82, 1977.
- [van Heesch] Dimitri van Heesch, "Doxygen", (<http://www.stack.nl/~dimitri/doxygen/>).
- [van Leitner2003] Felix van Leitner, "Exploiting SIMD Intruction", 2003.
- [van Lent1999] Michael van Lent, John E. Laird, "Developing an Artificial Intelligence Engine", GDC, 1999.

- [Varanese2002] Alex Varanese, "Game Scripting Mastery", Course Technology PTR, 2002.
- [Vienneau1995] Robert L. Vienneau, Roy Senn, "A State of the Art Report: Software Design Methods", The Data & Analysis Center For Software (DACS), (<http://www.thedacs.com/techs/design/Design.Title.html>), 1995.
- [Virtools] Virtools, "Virtools Dev", Virtools, (<http://www.virtools.com/>).
- [Virtools_AI] Virtools_AI, "Virtools Artificial Intelligence Pack", Virtools SA, (http://www.virtools.com/solutions/products/virtools_aipack.asp).
- [Virtual-Terrain] Virtual-Terrain, "Virtual Terrain Project", (<http://www.vterrain.org/>).
- [Vlissides1996] John Vlissides, "Pattern Hatching - Seven Habits of Successful Pattern Writers", C++ Report, November-December, (<http://hillside.net/patterns/papers/7habits.html>), 1996.
- [Voas1998] Jeffrey M. Voas, "The Challenges of Using COTS Software in Component-Based Development", In IEEE Computer Journal, pp. 44-45, 1998.
- [Vrije] Vrije, "CAVEStudy", Vrije Universitat, (<http://www.cs.vu.nl/~renambot/vr/html/intro.htm>).
- [VTune2004] VTune, "VTune", Intel Corp, 2004.
- [W3C] W3C, "W3C: World Wide Web Consortium", (<http://www.w3.org/>).
- [Walker2006] John Walker, "Technical Issues in Tools Development", GDC, 2006.
- [Walker2003] Matthew Walker, "Creating A 'Safe Sandbox' for Game Scripting", in Massively Multiplayer Game Development, Charles River Media, 2003.
- [Wall2"5] Michael Wall, "Maximizing Performance of PC Games on 64-bit Platforms", GDC, 2"5.
- [Wall2006] Michael Wall, "Optimizing Games for AMD Athlon 64 Processors in 2006 and beyond", GDC, 2006.
- [Walrath1999] Joshua Walrath, "30 Frames per Second vs. 60 Frames per Second", (http://www.daniele.ch/school/30vs60/30vs60_3.html), 1999.
- [Ward2002] Mark Ward, "Computer games start thinking", BBC News, (http://news.bbc.co.uk/1/hi/in_depth/sci_tech/2000/dot_life/2225879.stm), 2002.
- [Watt2000] Alan Watt, Fabio Policarpo, "3D Games: Real-time Rendering and Software Technology", Addison-Wesley, 2000.
- [Web3D2004] Web3D, "VRML97 Functional specification and VRML97 External Authoring Interface", VRML Archives Web3D Consortium, (<http://www.web3d.org/x3d/vrml/index.html>), 2004.
- [West2005] Mike West, "Debug and Release", in Game Developer Magazine, CMP, October, 2005.
- [Wheeler] David A. Wheeler, "SLOccount: set of tools for counting physical Source Lines of Code (SLOC) in a large number of languages of a potentially large set of programs", (<http://www.dwheeler.com/sloccount/>).
- [Whigham2005] John Whigham, "The evolution of a game engine", in Develop Magazine, Intent Media Publishing, 2005.
- [White2001] Stephen White, "Postmortem: Naughty Dog's Jak and Daxter: the Precursor Legacy", Gamasutra, (http://www.gamasutra.com/features/20020710/white_01.htm), 2001.
- [Wihlidal2006] Graham Wihlidal, "Game Engine Toolset Development", Course Technology PTR, (<http://www.getdbook.com/>), 2006.
- [Wilson2003] K. Wilson, "Game Object Structure", GDC, 2003.
- [Wilson2005] Kyle Wilson, "A Streaming Bestiary", (<http://gamearchitect.net/Articles/StreamingBestiary.html>), 2005.
- [Wilson2006] Kyle Wilson, "Managing Concurrency: Latent Futures, Parallel Live", (<http://gamearchitect.net/Articles/ManagingConcurrency1.html>), 2006.
- [Wloka2003] Matthias Wloka, "Batch, Batch, Batch: What Does It Really Mean?" GDC, 2003.
- [Wolka2004] Matthias Wolka, "GPU-Assisted Rendering Techniques", GDC, 2004.
- [Wright2005] Will Wright, "The Future of Content", GDC, 2005.
- [Wu2006] David Wu, "Threading Full Auto Physics", GDC, 2006.
- [Yersin2005] Barbara Yersin, Jonathan Maïm, Pablo de Heras Ciechowski, Sebastien Schertenleib, Daniel Thalmann, "Steering a Virtual Crowd Based on a Semantically Augmented Navigation Graph", VCROWDS, 2005.
- [Yiskis-FSM2004] Eric Yiskis-FSM, "Finite-State Machine Scripting Language for Designers", in AI Game Programming Wisdom 2, Charles River Media, 2004.
- [Yiskis2004] E. Yiskis, "A subsumption architecture for character based games", In AI Game Programming Wisdom 2, Charles River Media, 2004.
- [Zarge2006] Jonathan Zarge, Richard Huddy, "Squeezing Performance out of your Game with ATI Developer Performance Tools and Optimization Techniques", GDC, 2006.
- [Zarge2004] Jonathan Zarge, "Normal Map Compression with ATI 3Dc", AGDC, AGDC, 2004.
- [Zimmer1996] Walter Zimmer, "Pattern Languages of Program Design: Chapter 18 Relationships Between Design Patterns", Addison-Wesley Professional, (<http://www.bell-labs.com/topic/books/PLoPD1/toc.html>), 1996.

Appendix A

Existing Platforms and Their Associated Standards

Software engineering is all about not reinventing the wheel, but reusing existing code whenever there is an opportunity. Over the years, different APIs, libraries, and standards for 3DRTS have been elaborated. They are extremely useful for programmers, which should use them for several reasons: they are well tested and provide common and efficient functionalities. Some of the most notorious APIs are the 3D APIs such as the OpenGL and DirectX APIs. The next sections will describe some well-known standards including the ones used in this thesis.

A.1 OpenGL

OpenGL is the industry standard for 3D graphic API. Derived from the old IrisGL (the old proprietary graphics library for the SGI Iris workstation), OpenGL was announced in 1992 as an open standard directed by the OpenGL ARB (Architecture Review Board). The ARB is an association of many companies that oversee and vote on the API modifications. The newest version of the API is OpenGL 2.0. Some advantages of the OpenGL API are that it is a true portable solution and runs on most platforms. This is possible as OpenGL API is separating the implementation of the API from the specific hardware. OpenGL relies on an intermediate layer to be implemented for each platform such as OpenGL drivers for Windows PC. This gives to the developers the ability to access windows and frame buffer without dealing with the hardware specific layer.

A.2 DirectX

The Microsoft equivalent of OpenGL is called Direct3D and is part of the DirectX SDK. DirectX is a collection of APIs for various aspects of multimedia programming, including, specifically, 3DRTS. DirectX features a suite of APIs built on top of Microsoft OS and is, by consequence, only available on Windows PCs and Microsoft home consoles such as the Xbox and Xbox 360. The no portability reduces the API usage to a smaller set of simulations. Nevertheless, the DirectX SDK is well documented and well supported by major DCC tools vendors. For instance, Discreet has recently chosen to use DirectX over OpenGL for their default 3DSMax renderer [Discreet]. DirectX enable developers the access to specialized hardware features without writing specific code. The technology was introduced in 1995 and is recognized as standard for multimedia applications developments on Windows powered platforms. Direct3D, which represent Microsoft counterpart to OpenGL features rich and high performance 3D graphic library. Contrary to the OpenGL ARB committee, the DirectX API is designed by a single company, making it more reactive to technological advances and tends to adopt as standard some features faster than with OpenGL. However, the portability of OpenGL and the relative open spirit of the OpenGL API remain its principal advantage over DirectX.

A.3 OpenGL ES

The OpenGL ES (OpenGL for Embedded Systems) initiative is an attempt to speed up the development of 2D and 3D graphics on embedded systems. It consists to provide subsets of desktop OpenGL implementations, by removing the limitations of providing full OpenGL implementations. In fact, many 3DRTS do not use all the functionalities provide by the OpenGL API. Moreover, the OpenGL ES initiative can react more accurately to technology advances due to his semi-private management. Since OpenGL ES is based on OpenGL, no new technologies are needed. This ensures synergy with, and a migration path to and from desktop OpenGL. Other benefits coming with OpenGL ES include a smaller footprint a seamless transition from software to hardware rendering pipeline as well as the capability to extend the API. The roadmap of OpenGL ES is separate into categories for fitting the diverse need and capability of embedded systems (see Figure A.1).

Using these, multiple versions enable the API to meet the graphics requirements from a wide range of 3D devices such as mobile phones or home consoles.

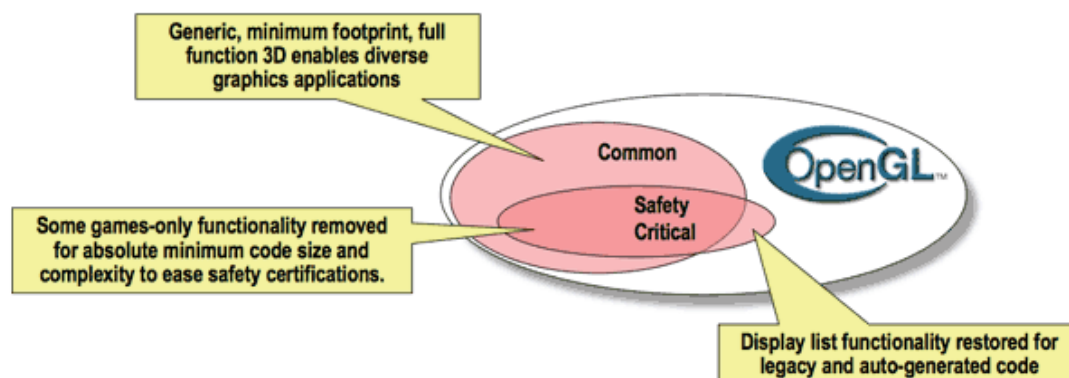


Figure A.1: The two OpenGL ES profiles: the Common Profile, and the Safety Critical Profile.

A.4 UML

The unified modeling language [UML] was introduced in 1997 as a combined effort from [Jacobson1999]. They were trying to provide analysis and design tools to validate object-oriented architecture design. UML is a notation that model objects and their interactions using visual representation such as diagrams. UML is an international standard directed by the Object Management Group [OMG], one of the largest consortium of software developers. The evolution of the UML standard and its methodology do not rely on particular architectures which implied that UML can be used by any software architectures design process [Fowler-UML1999]. The advantages of using such standard for describing objects interactions promote the ability to understand massive scale applications. Describing system architectures by adopting UML conventions such as collaboration, sequence and components diagrams greatly facilitate the communication between team members during the software design process. Naturally, UML methodology may not always be compliant with mission-critical software and their related hardware specific issues. Thus, UML may serve as a methodology to describe the high-layer of the system but should be adapted to mission-critical software needs.

A.5 XML

The Extensible Markup Language is a W3C [W3C] recommended general-purpose markup language capable for describing many different kinds of data. It is a simplified subset of SGML. Its primary purpose is to facilitate sharing data. Since developing XML-parser is a relative straightforward, many applications are intensively using XML files for describing application contents. This trend is also directly affecting 3D real-time systems developments, which use XML-based file system for sharing meta-data. Some key strengths of XML is that file remain human readable and its native ability to represent the most general computer science data structure such as records, lists and trees. The way that data can associate a structure with a field names with specific values help to instantiate applications in a convenient and efficient manners. However, XML also come with some weaknesses, which include verbose and redundant information, which lead to higher storage costs and higher bandwidth. This may be a problem on application running on hardware with restricted resources.

A.6 X3D

The X3D or Extensible 3D Modeling Language [W3C] is aiming at implementing an agnostic component. The main domains of usage of the X3D standard are closely related to Web3D applications. The X3D initiative is an evolution of the existing VRML97 standard [Carey1997]. X3D identified precisely types of extensions, legitimate structural dependencies, and behavioral interaction patterns [Parisi2003 823]. The X3D initiative tend also to encourage the data driven programming style as an actor for controlling the system

simulations relationship with interactive events. X3D also specific nodes and information in a human readable format based on XML. In contrast to metadata files systems such as Collada [Collada], X3D is intend to be use at run-time. X3D syntax was developed to be human friendly for representing data information. The drawback is the text-base nature of the files format. Thus, X3D files are not optimized for run-time performance. Finally, X3D targets exchange of readable information over the network using browsers or various DDC tools. From this perspective, X3D is not competing with the other formats, as its primary target remain interoperability and interchange of content, not run-time performance or metadata representations.

A.7 CONTRIGA

The CONTRIGA [Dachselt2002] project pursues the development of component-oriented 3DRTS. The CONTRIGA architecture relies entirely on structured documents describing the component implementation, their interfaces, and assembly configurations. The syntax is based on markup languages derive from XML syntax. CONTRIGA form an XML suite that contains multi-layered XML grammars and hierarchy as shown in Figure A.2. The lowest level in the hierarchy represents the scene graph using similar semantic than in the X3D standard. The semantics can be mapped for actual scene graph formats such as Java3D or VRML. CONTRIGA enforce the separation between the geometry and object behaviors, which provide an abstraction layer with proprietary files formats. The second layer in the hierarchy is the scene components. It encapsulates the scene graph semantics allowing hiding or combined scene graph attributes to form higher-level elements. On top of the hierarchy, CONTRIGA describe the scene layer, which represent the higher level of configuration. It represents a declarative description of a 3DRTS based on assembled component descriptions but also serves as an exchange format for 3D modeling packages.

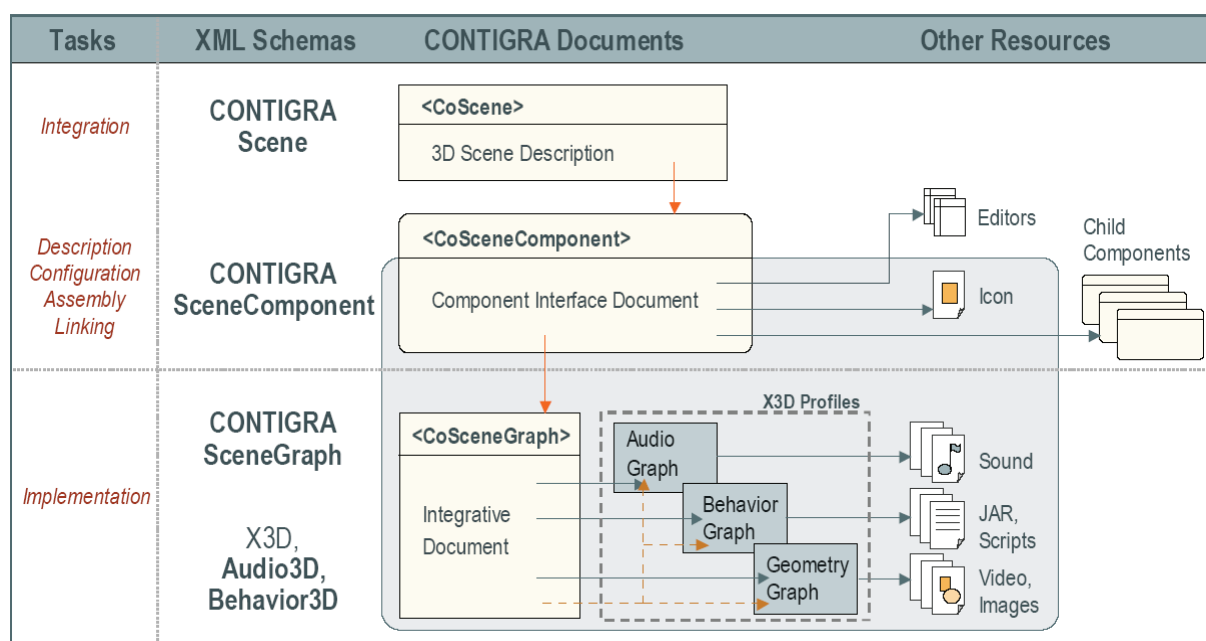


Figure A.2: The three layers of hierarchy found in the CONTRIGA standard.

A.8 XDS

The XDS file standard of eXtensible Data Stream [Sudden-Presence2004] represents a format, which is a generic, extensible, portable data format that supports most of the features of XML in a more compact representation. The idea is to combine the flexibility and readability of XML files but to limit the memory footprint. The XDS standard directly supports all C language types except for unions. The representation of XDS is two fold: a text based files comprised of a header followed by any number of records and a binary model for runtime usage. XDS stream used a two bytes value that define the class of the record while the item may contain expected alphanumeric and characters symbols. Moreover, by taking advantages of the real type

knowledge provide by the XDS stream allow more efficient transliteration than standard XML files. It result that the text files are generally using only one quarter of their original XML files size.

A.9 Collada

Collada [Collada] is a collaborative design activity for establishing an open standard Digital Asset Schema for 3DRTS. Collada is not a delivery format, but rather is an interchangeable format for 3D assets. At the beginning of this initiative, SCEA was trying to design a unique centerpiece of digital asset tool chains use by the 3D interactive industry for helping the development of applications on Sony hardware [Stenson2006]. So far, the major DCC tools companies as well as some middleware software and hardware manufacturers support the Collada initiative. Many advantages can come up from this initiative. It may reduce the development of specific exporters for each DCC and avoiding to continuously updating converters and plug-ins at each software update. This also helps developers to be less restricted to one particular tool, as the Collada files format guarantees to encode the exact information. Collada is not using proprietary format and is not restricted to 3D content. Current and future version of Collada will support features such as physics, shader effects files, or sound objects. The concrete integration of Collada within the content creation pipeline is depicting in Figure A.3. In this configuration, artists are creating content within a supported DCC tool that can export Collada XML data files. Those files are then handled by the end-user application through Collada API and integration template. Depending on the architecture design, this integration can occurs as preprocessing steps for converting Collada files to proprietary files or directly at run-time for creating simulation objects.

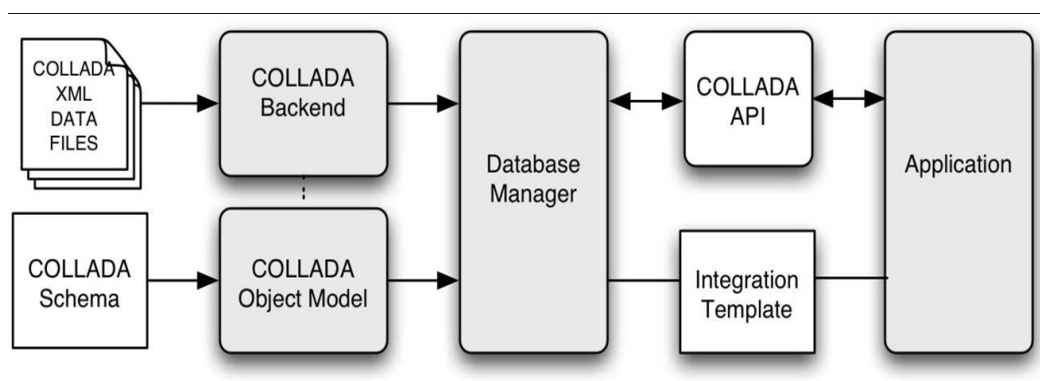


Figure A.3: Collada integration pipeline.

A.10 H-ANIM

The H-ANIM standard [HANIM] specifies a common way of representing humanoids and their associated skeleton generally for VRML97 files format. The human body consists of a number of ‘Segments’ (such as the forearm hand and foot), which are connected to each other by ‘Joint’ (such as the elbow, wrist and ankle). The full H-ANIM hierarchy is composed of 94 skeleton joints and 12-skin segment including the head, hands, and feet. The H-ANIM standard defines a strict naming convention for the joint hierarchy. However, the standard does not mandate that all joints and segments are used, and thus allow to easily creating animation systems, which can rely on standard names from which joints, can be referenced. Thus, rotation can be set through events affecting individual joint. Every joint will modify the posture of the body and the usage of a global root joint described as *HumanoidRoot* serve for rotation and positioning the avatar in the 3D world. At its creation, the specifications restrained a specific number of bones as an upper limit. If this number still suffices for most applications, including crowd simulations, it may not suffice for all occurrences. For instance, characters animations systems such as the one used in the Source Engine [Valve-Source2004] and the Unreal Engine 3.0 [Epic-Unreal] may use few hundred bones for creating complex facial animations systems. Another limitation of the H-ANIM standard is that the animation specifications are focus on virtual human animations (see Figure A.4). Thus, the H-ANIM standard will not perform well for customized skeleton meshes such as animals, vehicles, or monsters.

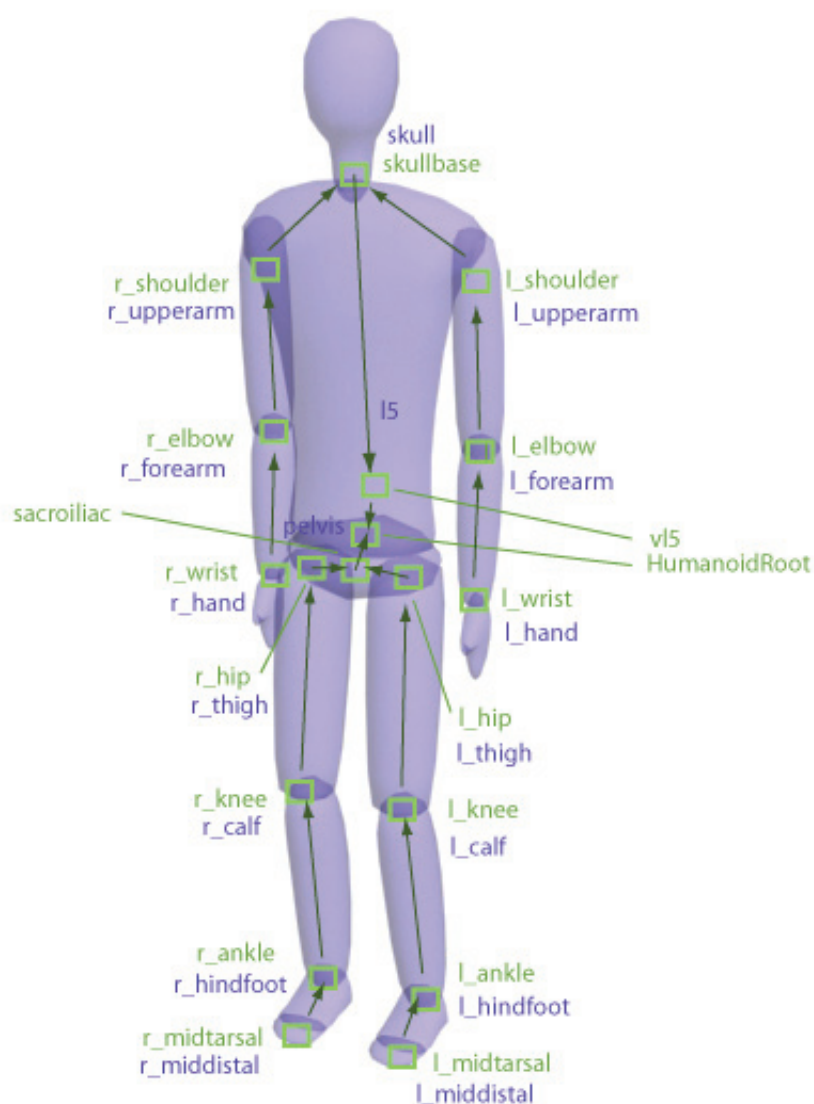


Figure A.4: Subset of the H-ANIM standard commonly used by H-ANIM compliant characters.

A.11 XNA

For managing the constant increase of software complexity for 3DRTS, Microsoft elaborates the XNA initiative, which tends to embed several cross-platforms technologies especially developed to ease 3DRTS developments. XNA was introduced during the Game Developers Conference back in 2004 [Allard2004]. In 2005, Microsoft took the opportunity to provide additional information concerning the concepts and tools forming the XNA software platform. Fully component oriented game engine architectures are yet to come. However, the transition may be faster than expected, especially in the light of the very recent XNA initiative from Microsoft [Microsoft-XNA]. XNA target to promote components based architectures for next generation of hardware. XNA promote system architectures that integrate software developments and content creation pipelines. It try to prevent the current unfavorable “80/20” ratio (80% time being spend on construction and integration of technologies, 20% time spend on creation and development of product specific technologies). XNA initiatives want to participate in maturing systems architectures for 3DRTS, by evolving from features-driven software toward architecture-driven engineering. Currently, the number of existing solutions for both the academic world and the gaming industry is overwhelming. The wide range of toolkits, engines, and development environments are too different for providing some standardization for completing such software developments. The XNA initiative clearly intend to combined game engines with a set of comprehensive suite

of production tools covering both software developments and content productions processes. Microsoft point that from the list of more than 300 existing solutions [Isakovic] are falling both into open source, free, and commercial categories, classified according to functional features being supported. Presently, this type of features-driven classification is predominant. However, we may expect that soon not functional but rather architectural features will start to play the most important role in classification and selection, as endorsed by the XNA initiative. Furthermore the XNA Studio which extend the Microsoft Visual Studio 2005 Team System software [MS_Team_System2005], with additional features toward the specific element such as team-based development environments tailored for 3DRTS. XNA was intently built on top of existing and well-proven tools of choice from the development community. It adds assets management with defect tracking systems and a batch of converters and automates tools, which were so far; develop in-house by the different developers. By regrouping all theses tools altogether within consistent APIs and GUIs provide a more streamlined development environment. Moreover, Microsoft understands that the massive scale of current applications cannot anymore be developed with the programmer centric scheme. By consequent, XNA is design to fully embrace all the different people involve including artists, designers, QA testers producers and naturally programmers. By allowing developers to concentrate on the key elements in the creation process rather than on code and tools portability, is the main goal of this initiative. For instance, the management of simulations assets is completely integrated within XNA helping the transition to more dense and complex data managements [Keller2006]. Nowadays, XNA is widely spread in the gaming industry and especially by all software develop at Microsoft Game Studios [MS-GS].

A.12 OpenInventor

OpenInventor is an object-oriented 3D toolkit offering a comprehensive solution to interactive graphics programming problems. It presents a programming model based on a 3D scene database that dramatically simplifies graphics programming. It includes a rich set of objects such as cubes, polygons, text, materials, cameras, lights, trackballs, handle boxes, 3D viewers, and editors that speed up your programming time and extend your 3D programming capabilities. Built on top of the OpenGL API, it defines a standard file format for 3D data interchange. It introduce simple events model for 3D interactions and provide set of functionalities such as object picking. OpenInventor is a cross-platforms solution for 3D graphics developments that encourages programmers to create new customized objects. Widely used for CAD and within academic research, OpenInventor is a mature toolkit even so it does not support latest 3D graphic functionalities such as advances shaders rendering. After many request from the GNU/Linux community, SGI has open sourcing the OpenInventor toolkit allowing a larger community to enhance it.

A.13 OpenSG

OpenSG [BMBF] is a portable scenegraph system to create real-time graphics programs, e.g. for virtual reality applications. It may not be a perfect match for all type of 3DRTS, but some elements and design idea are worthwhile for closer study and adaptation. OpenSG API consists of a library of routines that are necessary for 3D graphics simulations, especially with the interconnection of VR hardware. OpenSG is a project funded by the German Ministry for Research and Education. OpenSG define architecture and data structures together and provide important components based on four modules as depict in Figure A.5

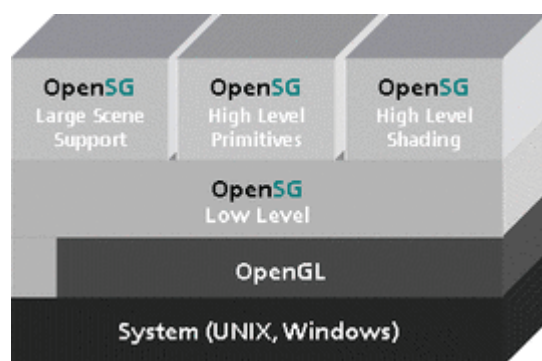


Figure A.5: The OpenSG is built around four modules separating low levels and high level of interactions.

A.14 OpenSceneGraph

OpenSceneGraph [Osfield] is an open source cross-platforms toolkit for developing high performance 3D graphics applications, such as visual simulations, games, virtual reality and augmented reality systems. The API is written in standard C++ on top of the OpenGL API. It runs on many platforms including all Windows platforms, GNU/Linux, IRIX, Solaris, and FreeBSD OS. OSG was successfully port for gaming systems such as the Microsoft Xbox and Sony Playstation 2. Based on a scene graph, it provided an object oriented framework, freeing developers from implementing and optimizing low-level graphics calls and provides many additional utilities for rapid development of graphic applications. The OpenSceneGraph initiative makes the benefits of scene graph technologies freely available. It makes full use of advanced C++ programming concepts such as STL and Design Patterns. The key strengths of OpenSceneGraph are its performance, scalability, and portability. OpenSceneGraph scope goes beyond the simple scene graph representation and offer a set of functionalities to improve the rendering performance. OSG is the principal 3D renderer use in this thesis for several reasons: OSG support natively many rendering optimizations such as view frustum culling, occlusion culling and early removal of objects. The scene graph also integrates nodes for handling LODs, as well as organizing the scene base on cost functions such as states sorting. Many researchers used this toolkit and refer that its performance surpasses many existing scene graph toolkits such as Performer [SGI], VTree [Quantum3D], NVSG [NVIDIA-NVSG] or Java3D [Sun-Java3D]. One clear advantage is that OpenSceneGraph offer the mechanisms to extend the scenegraph with customized node such as crowd rendering or infinite terrain [Demeter, Virtual-Terrain, Thatcher2002]. Being supported by 3D hardware vendors such as 3DLabs [3DLabs], the system always encapsulates most OpenGL functionalities but also benefit from additional plug-ins extending the system capabilities. Such plug-in include support for physics or sound [Backman] nodes. Beside these nodes, it also supports more than 45 files formats such as Collada, OpenFlight, and TerraPage (see Figure A.6). Using OpenSceneGraph allow to concentrate on content management rather than on low-level coding. In addition, OpenSceneGraph is a very portable toolkit that includes the support for IRIX, Linux, Windows, Xbox, and Playstation 2.

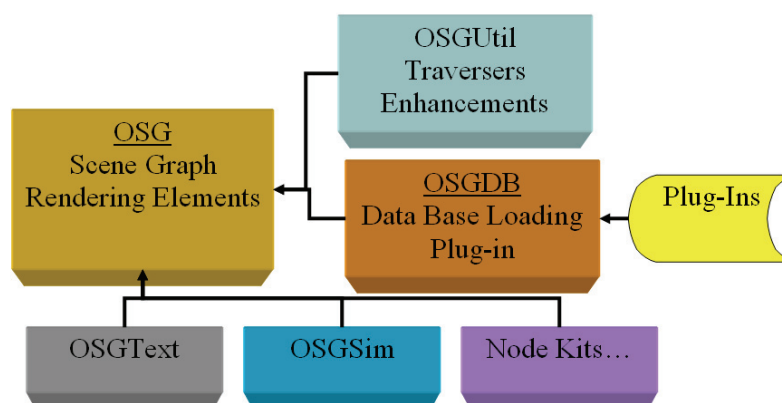


Figure A.6: OpenSceneGraph Plug-in encapsulation.

A.15 OpenAL

OpenAL [Creative-OpenAL] is the open audio library that aims to become the cross-platforms 3D audio standard for gaming applications and many other types of audio applications. Albeit relatively recent, OpenAL has working implementations on most platforms that are interesting to 3DRTS developers, and has thus become a very attractive alternative for handling 3D sound.

A.16 FMod

FMod [Firelight] is a cross-platforms audio library supporting the similar functionalities on a wide range of hardware. The API include support for software sound mixing, 3D audio, music playback (MP3, Ogg Vorbis, MID support, etc), as well as more advance 3D sound processing such as Dolby support. The FMod API is free for non-commercial usage.

A.17 ODE

The Open Dynamics Engine [Smith] is an Open Source, high performance library for simulating rigid body dynamics. It is fully featured, stable, mature and platform independent with a light C/C++ API. It has advanced joint types and integrated collision detection with friction. ODE is useful for simulating vehicles, objects, and virtual creatures in virtual reality environments. It is currently used in many computer games, 3D authoring tools, and simulation tools. ODE is the more mature library for physics computation under the LGPL license.

A.18 CEGUI

The Crazy Eddie's GUI System [Turner] is an Open Source library providing windowing and widgets for graphics APIs/engines where such functionality is not natively available, or severely lacking. The library is object orientated, written in C++, and targeted at 3DRTS developers to reduce the time spend on building GUI sub-systems as shown in Figure A.7. CEGUI present an OO architectures for describing user interfaces, and supports events. It has already been integrated for use in different 3D engines such as Ogre3D [Streeting], Irrlicht [Gebhardt], Delta3D [Delta3D_Dev_Team] and Nel [Nevrax]. CEGUI is licensed under the LGPL License.



Figure A.7: CEGUI widgets are independent of the underlined 3D API. It can work with existing 3D engines and pure OpenGL or DirectX applications.

A.19 Doxygen

Doxygen [van Heesch] is a documentation system for C++, C, Java, Objective-C, and IDL. It is inspired mainly from JavaDoc. Doxygen can be combined with different tools for generated documentation in compressed html or latex for instance. It may also generate class and integration [Sun-JavaDoc] diagram as well as providing the ability to integrate documents such as white paper directly into the generated documentation.

Appendix B

Hardware Architecture for 3DRTS

The wide range of hardware dedicated for 3DRTS developments, are continuously promoting new technical approaches for improving the processing throughput [Reimer2005]. The hardware repartition can be separate into different categories:

- **Mobile Systems:** this category includes mobile phones or small-embedded devices that require low power consumption. They come with limited 3D graphic chipsets due to low consumptions requirements and screen size.
- **Portable Systems:** this category includes systems such as handheld consoles, PDA and Pocket PC. Those systems are incorporating previous desktop chipsets, which have been miniaturized. The advantages of reusing the technology facilitate the developments and adaptations of software. For instance, the Sony PSP, which was release in late 2004, is sharing many similarities with the Sony Playstation 2. The approximate 5 years that separate the release of these two systems have made possible the integration of desktop technology into a handheld device. Thus, portable systems are the results of adopting older desktop technology with a specific attention to power consumption.
- **Desktop Systems:** the category includes both desktop computers like PCs but also embedded systems such as home consoles. These systems use the more technical advanced coprocessors within a specific budget. Most of those systems are clearly focus on optimizing processing throughput with little consideration for power consumption or heat issues. The average power consumption can reached more than 250 Watts.

B.1 Evolution of multi-core CPUs

The Figure **B.1** describes the evolution of multi-core CPUs. As we can observe, processors architecture are moving from a single-processor design toward a multi-core processors where the number of cores remain reasonable and often present a symmetrical approach. This ease the development of scalar and parallel application and promote and continuous increase of CPU performance. The next step would be the incorporation of hardware design embedding a large number of specialized and assymetroc processors refers as the many-core era. This large increase in the number of hardware threads will require applications to be massively parallelized.

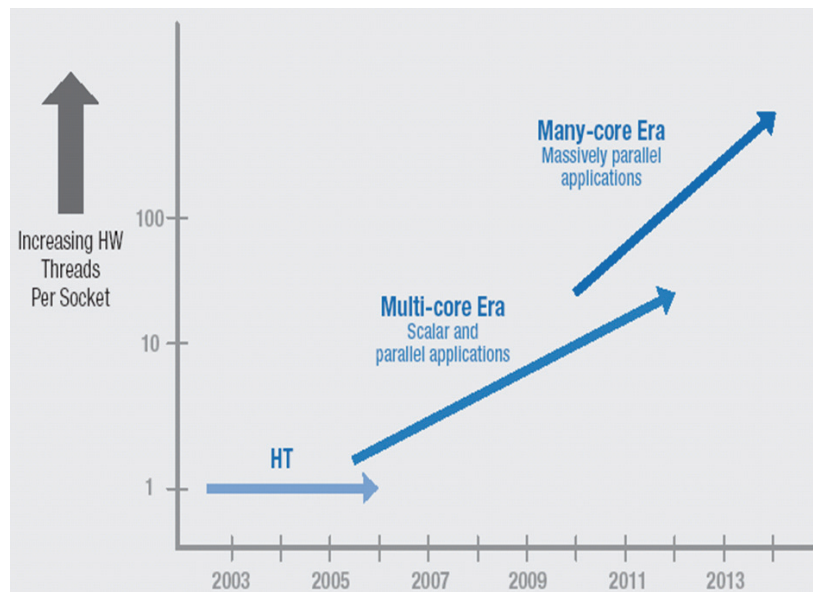


Figure **B.1**: Evolution of Multi-core CPUs [Plumb2006].

B.2 Mainstream PCs and PC Workstations

The mainstream PC and PC workstations CPUs market is disputed by two major manufacturers [AMD, Intel]. The Figure B.2 describes the dual-core architecture proposed by AMD. In this configuration, we can see that both cores have their dedicated level-2 cache memory and that an intermediate request interface serves for dispatching the tasks.

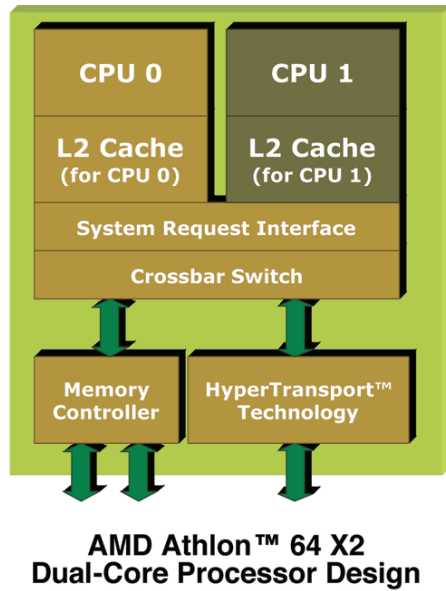


Figure B.2: The dual-core architecture design used by AMD for its 64-bits X86 processors.

The proposed roadmap for CPU evolution from AMD is describing the evolution of this design for more than two cores. In fact, AMD and Intel are planning to provide quad-core CPUs for mainstream PCs in 2007. Intel is also promoting the usage of multi-core CPUs with a different internal approach. This leads to the propagation of multi-core PCs as shown in Figure B.3 [Intel2005]

	2004	2005	2006*
Desktop Performance	65% HT	Shipping dual core	>70% dual core
Servers	100% HT	Shipping dual core	>85% dual/multi-core
Mobile Performance	Mobile Optimized Micro-Arch	Shipping dual core	>70% dual core

Figure B.3: Projection of tie-ratio of PCs with multi-core CPUs.

B.3 Hyper-Threading

Intel introduced the concept behind the Hyper-Threading technology during the 2002 Intel Developers Forum conference. The Hyper-Threading technology was the first exposure for mainstream computers of a research work done at the University of Washington with the Simultaneous Multithreading Project [Egger1995]. From a historical perspective, superscalar processors were capable of trying to squeeze extra performance for single-threaded application. The introduction of out-of-order execution and long pipeline were also providing better performance on serial code stream. However, the difficulties for compilers and hardware optimizations to automate generation of parallel code on an instructions level were reaching a plateau. The incremental performance increase was not justifying the architecture complexity. Thus, for improving performance throughput, other alternatives were considered. The researches made with the Simultaneous Multithreading

project bring the idea of providing parallelism on a thread level for single core processor. Its means that a single processor can have two or more execution threads running simultaneously. Hyper-Threading technology allows multi-threading applications to avoid performance problems related to blocking operations. From hardware perspectives, the processor architecture has to provide several execution units, which enable some instructions to be executed in parallel. This concept was also used for out-of-order execution models, where the two branches of a if condition can be executed on different pipelines by simply discarding the wrong branch. These look-ahead facilitate flow optimizations of executed instructions. The concrete implementation of Hyper-Threading technology relies on allowing executing two instructions within the same clock cycle as long as they do not use the same execution unit. On Intel implementation, the two-hardware threads shared the same level-2 cache memory but hat their own dedicated level-1 cache memory (see Figure [Figure B.4](#)). The processor then schedules the instructions execution depending on which units are available. From a high-level perspective, the two hardware threads looks like two separate processors. In the fact, the two threads will not run at full speed as some processing units are still shared. In general-purpose applications, a speed-up of 5-20% can be achieved.

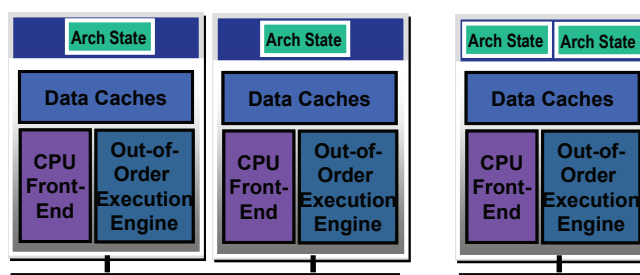


Figure [B.4](#): The left part represent the Hyper-Threading architecture design used by Intel in its Pentium IV line of processors. On the right, the same processor without Hyper-Threading.

B.4 XCPU

The XCPU chip contains three processor cores that shared 1MB L2 cache and the crossbar linking the CPUs and L2, plus clocks, test and debug circuits and I/O as described in cache [Figure B.5](#). The XCPU is a three-way symmetric multiprocessors; each core is identical. The CPU clocks at 3.2GHz, and has a maximum front side bus speed of 21.6GHz per second. IBM designers also built in substantial clock gating, to try to minimize power consumption. The cores are built around an enhanced version of the Power PC architecture. Perhaps most interesting is the VMX section. "VMX" is IBM's moniker for their SIMD (single-instruction, multiple data) block, akin to Intel's SSE instructions. The standard PowerPC has 32 VMX registers, but the designers expanded the VMX registers to 128. There is also a standard floating-point unit, but it is likely that the VMX unit will handle most of the floating point. Every VMX unit can handle two simultaneous threads, and IBM added some instructions specific to 3DRTS and streaming media. For example, there are now Direct3D-specific pack and unpack instructions and floating-point dot product instructions. The CPU itself has sixteen 32-bit performance counters, which tie into the Xbox 360 developer kit tools, and enables fine-grain performance analysis of Xbox 360 applications code. One of the key elements in the XCPU design relies on the ability to stream data at high bandwidths. For reads, the CPU can bypass the L2 cache, prefetching data directly into the L1 cache. Each of the three-processor cores can have eight loads or prefetches outstanding. When the CPU writes back data, the CPU can write through the L1 cache, and portions of the L2 cache can be allocated as locked buffers just for streaming data. The GPU can read directly from the CPU L2 cache in 128 bytes wide increments. The L2 cache size is 1MB, and is shared among the three-processor cores. Data stores clock at the full 3.2GHz of the CPU, while the control logic and buffers run at half the CPU clock, or 1.6GHz. As the symmetric, multi-core design, applications built on this processor have to be performing with being smart by using intensively threads.

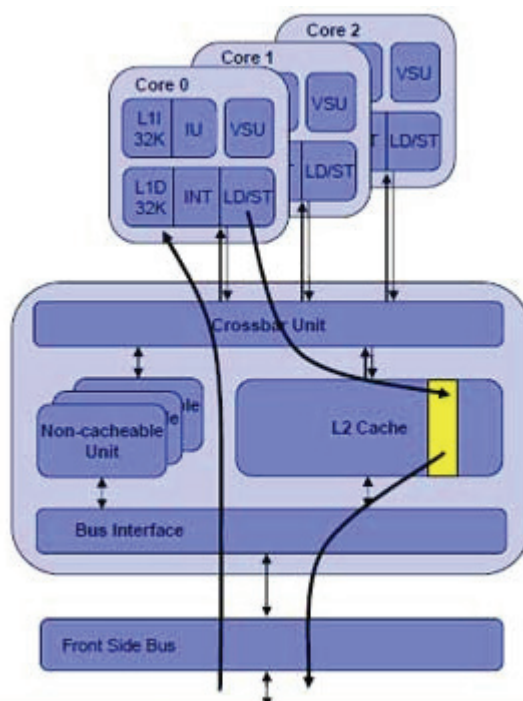


Figure B.5: The XCpu communications layer between the three cores unit and the crossbar unit.

B.5 CELL

The CELL processor is born from the collaboration of Sony as a content provider and IBM as a leading-edge technology and server company and Toshiba, as a development and high-volume manufacturing technology partner. This led to high-level architectural discussions among the three companies during the summer of 2000. The Cell objectives were to achieve 100 times the PlayStation 2 performance and lead the way for the future. During this interaction, a wide variety of multi-core proposals were discussed, ranging from conventional chip multiprocessors to dataflow-oriented multiprocessors. By the end of 2000 an architectural concept had been agreed on that combined the 64-bit Power Architecture with memory flow control and “synergistic” processors in order to provide the required computational density and power efficiency (see Figure B.6). The objectives were to provide extreme performance for 3DRTS, by avoiding performance limitations imposed by memory latency and bandwidths [Deloura2005]. The CELL architecture combine a dual-threaded, dual-issue, 64-bit Power-Architecture compliant Power processor element (PPE) with seven synergistic processor elements (SPEs). The CELL processor use a SIMD organization for both PPE and SPEs units dedicate for applications that have to process large datasets on small instructions set such as 3DRTS. The PPE is a 64-bits processor that does not dynamically reorder instructions at issue time (e.g., “in-order issue”). Synergistic processing elements are instructions-set architecture optimized. The SPEs operate on a local stored memory (256 KB) that stores instructions and data. The CELL processor programming model emphasis on streaming models by supporting fast message passing from the PPE and other SPEs. Another element is related to shared-memory multiprocessor model. The CELL processor can process tasks with a cache-coherent and shared-memory programming model. For the SPE units, all DMA operations are cache-coherent. One particularity is the asymmetric thread runtime model offered by the CELL processor. Threads can run on either PPE or SPEs and can interact with each other like in a conventional SMP system. This model provides an excellent foundation for many CELL programming models by extending the thread or lightweight tasks models of a modern OS to include processing units having different instructions sets such as the PPE and SPE. Various scheduling policies can then be applied to both the PPE and SPE threads to optimize performance and utilization. The synergistic processors in CELL provide a highly deterministic operating environment. Since they do not use caches, cache misses do not factor into the performance. In addition, since the pipeline scheduling rules are quite simple, it is easy to statically determine the code performance. Even though the SPEs operate on local memory, the DMA operations are coherent in the system, and CELL is a nine-way SMP from a coherency perspective. Ease of programming was also the primary motivation for including the Power

Architecture SIMD extensions on the PPE. This allows for a staged approach, where code is developed, and then SIMD-vectorized in a familiar environment, before performance is enhanced by using the synergistic processors.

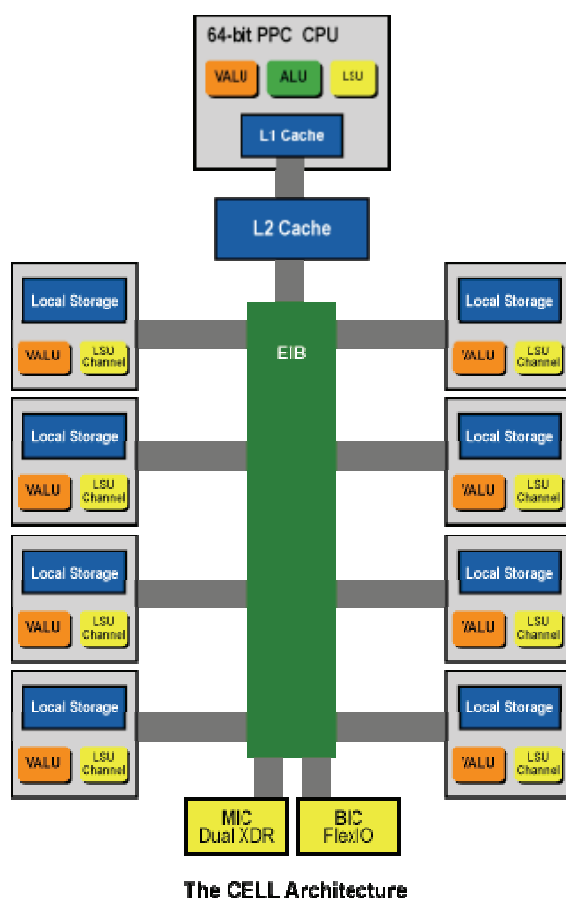


Figure B.6: The CELL architecture featuring seven active SPU units and one PPU unit.

B.6 GPU Architecture

The rapid evolution of 3D chipsets hardware for 3D real-time rendering and the introduction of shaders' languages have offered more options for creating 3D effects improving scenes realism. Current GPUs are constituted of several components. The two most important elements are the vertex and pixel pipeline. From a global perspective, the management of vertices, primitives, tessellation, vertex pipeline, clipping, and culling are part of the geometry management. On the other hand, the pixel pipeline, depth and alpha tests as well as fog and alpha blending are part of the rendering. With the current scheme, each component modifies the input and communicates the output to the next component as describe in Figure B.7. This GPU pipelined architecture is going to be modified with the introduction of DirectX 10. The new API is introducing a paradigm change, notably with the introduction of unified shaders' architectures [Glassenberg2006, Myers2006, Riguer2006]. In fact, the Xenos GPU that is incorporated into Microsoft Xbox 360 home console presents unified shader architecture [Doggett2005].

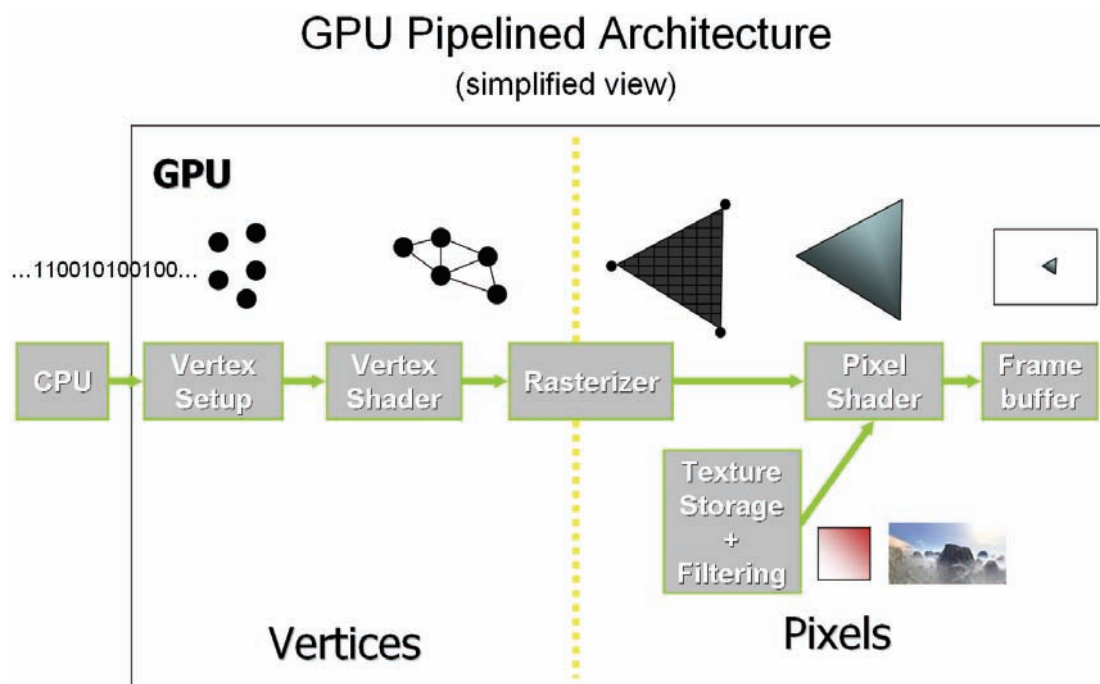


Figure B.8: GPU pipelined architecture [Kiel2006].

B.7 Physic Processing Unit (PPU) Architecture

Recently, a new kind of hardware dedicated for 3DRTS have emerged. It consists to provide specific hardware multi-core architectures tailored for the particular workflow and data types associated with the geometric and linear algebraic calculation for physics [Ageia2006]. The Figure B.9 describes the architecture retained by [Ageia] for their PhysX hardware.

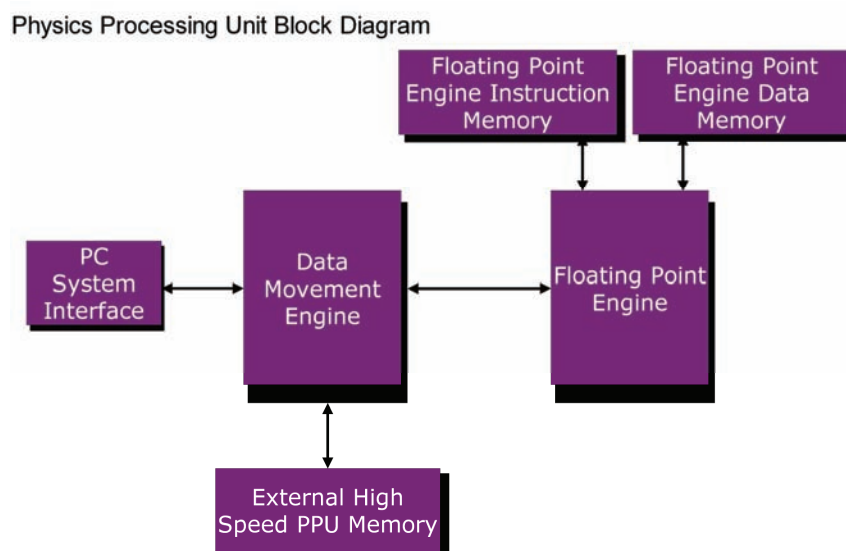


Figure B.10: PPU Block Diagram.

Appendix C

Cross-Platforms Considerations

The early interactive simulations were not intended to run across different platforms. Some reasons were that no software or hardware standard was clearly defined but also related to the scope of these projects. In the general purpose, they were either limited to run on predefined hardware and the project complexity and size was usually small. Thus, developers were rewriting modules to fit each platform on demands. They were trying to reuse simulation code, but very little from the system itself. The hardware capabilities were not encouraging to produce reusable architectures, as many critical code sections had to be written in machine specific assembly languages. Every hardware come with its set of functionalities, development teams were generally organized so that programmers become expert for one single platform. However, with the increase complexity and costs involved, this technical approach become less a viable option. Software becomes quickly deprecated. Software publishers promote products using short-term marketing campaigns. Having dedicated teams working on each specific platform would increase this period by months. As an outcome, cross-platforms software are developed in parallel, which require by definition a better architecture that can shared most of the code base [Malenfant2000]. The impact of cross-platforms developments involves more people than just programmers. It means that the installed base of potential customers is naturally bigger than single platform product. It helps to recover from investments. Moreover, cross-platforms software developments produce more robust and reusable code as applications have to run and react in a same way in various configurations. For instance, it is not possible to rely on internal side effects of hardware specific APIs. In addition, different compilers will produce different sets of problematic and by producing a code base that fulfill them oblige developers to produce more restrictive and portable code. This leads to adopt software engineering practices that rely on modularity. Many individual low-levels layers interact directly with the hardware within self-contain modules. Each of those modules has to present similar interfaces, while optimizing the computation for each platform. Another problematic related to cross-platforms developments is directly affecting the content creation. Content exploitations need to be adapted for each platform. In general, cross-platforms developments for 3DRTS use one platform, which serve as a reference model. This platform often refers to the lead platform is generally chosen for the following reasons:

- *Installed base*: The lead platform is chosen base on the platform with the higher sale potential.
- *Ease of development*: The lead platform is chosen base on the maturation of the tools and APIs coming with its Software Kit Unit (SKU).
- *Low-Level Denominator*: The lead platform is chosen base on hardware specifications. It is assume that if an application can run on lower technology hardware, it will also run on higher systems.
- *Business Choice*: The lead platform is chosen base on business agreements the publisher have made with hardware manufacturers such as limited exclusivity.

Finally, developing cross-platforms software help to understand the ramifications of each system, and some problems that may be hid by one platform can be discovered. Such problems include cache management and content creation [Carter2002]. Computer technologies evolve constantly and as a correlation, development time and costs are rising [Daglow2005]. Providing a system capable of running different simulations on several platforms is critical. In fact, bad managements may force long developments to switch their targeted platforms. This affect also applications which were first intended to run on a single platform including PCs, which are slowly moving from 32-bits to 64-bits OS. 3DRTS applications are clearly directly concerned, especially the one targeting moving platform such as mobile phones, where the wide range of hardware and their rapid evolutions make hardware predictions very difficult.

C.1 Component-Based System and Cross-Platforms Projects

CBD enforce cross-platforms projects by offering abstraction layers on top of the hardware-dependent components. This gives the opportunity to keep platform dependent data types confined [Goodwin2005]. The Figure **Figure C.1** illustrates this hierarchy where the main systems components directly use by clients are cross-platforms. This may lead to create interfaces for renderer and sound manager. Then, every platform will

provide a concrete component implementation, which communicate with the hardware. However, some components (in our example the AI engine) can be fully hardware-independent components.

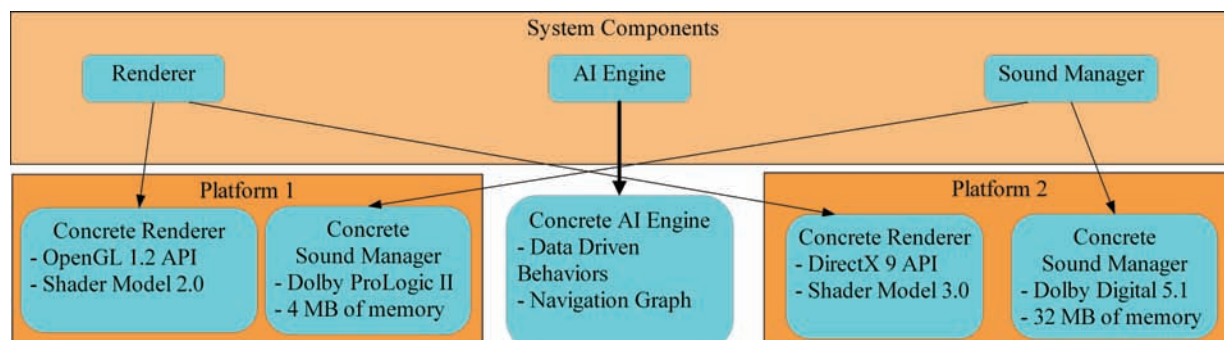


Figure C.1: System components and its underlined platform specific implementations.

C.2 Content Creation Pipeline and Cross-Platforms Projects

The content creation pipeline needs to export platforms specific data carefully. To clearly separate the system from DCC tools providers, the conversion is consequently written to export the content from DCC tools into a metadata files format. The Metadata files format can be manipulated using an editor that will provide the required information for generating the platform specific data. The specificities include files organizations on the disk as well as the data representation themselves [Bilas2003, Denman2003, Champoux2006]. Assets are adapted with appropriate geometry and textures complexity. The usage of meta-language serving as an intermediate step between DCC tools and run-time engine is becoming a standard feature [Collada, Criterion, Microsoft-XNA, Stenson2006]. They free developers from being too heavily dependent on external software such as DCC tools providers. Another strong advantage with metadata files format is the ability to export contents that are later adapted to fit hardware limitations. This increases the system flexibility. Let us imagine that an application can display and control four characters interacting within an environment. To improve the simulation, developers expect to optimize the resources usage in such ways that characters and environments fit the memory footprint and geometry complexity of the targeted platform. Now, if we assume that simulation designers consider extending the simulation with another character, a problematic may emerge. If the system cannot scale down 3D models, it will prevent the opportunity to adjust the simulation later. Generally, the interactions mechanisms are configured at the very end of a project. Allowing designers to modify simulations behaviors is beneficial. Therefore, if the system rely on metadata files, it becomes possible to regenerate the 3D models with extra parameters. In our situation, the character geometry complexity may be reduced to keep the same workflow. The iterative process involved when designing 3DRTS requires being able to adapt data on demand. This is particularly reflected with embedded systems such as game consoles. These platforms are fine balanced systems [LaMothe2005]. They may be lacking some performance in one area but compensate in another one. Therefore, a platform supposedly inferior may still perform some tasks more efficiency than another one. Successful cross-platforms architectures are not defined only by the number of platforms supported but also by the system capabilities to optimize and balance the workflow for each platform.

C.3 Metadata Files

The idea behind metadata files is to describe contents and information in a platform independent form. Metadata files can be exposed as convenient files formats rather than compiled into optimized and platform dependent binary files systems. Generally, metadata files are not use at run-time, so they can maintain a good level of clarity. Metadata files serve as a bridge between the engine and the content creation pipeline by providing a data abstraction layer allowing developing dedicated tools. This platform independence data representation, allow storing all the information describing the assets. Eventually, metadata files can provide additional information such as scaling methods as well as describing generic platform specificities such as suitable screen resolutions or system memory. Another strong argument for metadata files formats is that they promote the use of sanity-check procedures as well as keeping content managements and deployments more synchronized. This is important as during development, modifications made on the code or content level may

have an impact on particular platforms. When dedicated team members work on a specific platform, being able to discover them upfront is beneficial.

C.4 Loading Time Consideration

Metadata files can be used during the development process for loading assets. However, they are not optimized for loading the information in an efficient way. The structure of metadata files relies on ASCII file formats such as XML that need to be associated to a file parser. Unfortunately, most storage disks such as hard drives and optical disk drives are relatively slow to access information [Denman2003]. This affects the way simulations behaved with loading assets. Slow loading time may penalized the system and distract end-users, which will eventually not continue the experience.

The evolutions of the diverse hardware components that form a computer do not evolve in parallel with the same speed-up improvements. If CPUs and GPUs performance double every 12-18 months, the memory bandwidth improvement is following a slower path. As an analogy, the transfer rate of storage devices such as hard drives and optical disk drives does not scale with the increase of disposal memory. As an outcome, the time to feed the memory with data is increasing with time. This requires handling the data loading using compressed data and streaming algorithms [Wilson2005]. The Table C.1 describes the time to fill the available memory using different storage devices assuming no compression is used.

Table C.1: Loading time.

<i>Name</i>	<i>Year</i>	<i>Memory Size</i>	<i>Average Seek Time</i>	<i>Average Bandwidth</i>	<i>Time to Fill</i>
HDD 5400rpm, 8MB, 40GB UDMA/100	2003	256MB	15.6ms	25.1MB/s	10.21s
HDD 7200rpm, 16MB, 500GB, S-ATA2	2006	1024MB	8.5ms	48MB/s	21.34s
CD-ROM 32X	2000	64MB	114ms	2.8MB/s	22.97s
DVD-ROM 12X	2006	512MB	115ms	11MB/s	46.66s
Blue-Ray 2X	2006	512MB	? (>115ms)	9MB/s	57s

C.5 Binary Files Formats

The loading performance must remain optimal to eliminate or reduce these overheads. This leads to store the information into memory friendly architectures. For instance, it is more efficient to load C++ objects directly using compression algorithms such as the LZO [Oberhumer] that rely on bytes [Champoux2006]. In effect, modern CPUs have enough processing power to decompress on the fly data. In addition, programmers need to realize that the memory model used by most computers architectures, force data to be aligned on X-bytes boundaries. Some systems either will suffer from a performance hit or will simply refuse to read data that are not aligned. The IBM PowerPC970, which is found in Apple G5 and next-generation of home consoles hardware is doing automate re-alignments on the information. This may produce wrong data when direct memory access is used. By default, most platforms will align data for best performance, which may end-up using more memory in some circumstances. For reducing this overhead, padding data structures can be very precious on memory-limited systems. For illustrating the problematic of memory usage with data type alignment, let us investigate the following example:

Listing Memory Padding:

```

struct Unpadded
{
    uint32 uiA;
    char cB;
    // 3 bits of padding
    uint32 uiC;
    // 5 bits of padding
    char cD[7];
    uint64 uiE;
    uint32 uiF;
};

// sizeof(Unpadded) == 36

```

Here, we assume that this small structure is compiled on a 64 bits processor requiring data types to be aligned. As depict in Figure C.2, the unpadded structure required 36 bytes. Without the extra bytes taken by the alignments, the structure would require only 28 bytes. We can decrease this overhead by reorganizing the data as shown in the next listing.

Listing: Padded

```

struct Padded
{
    uint32 uiA;
    uint32 uiC;
    char cB;
    char cD[7];
    uint64 uiE;
    uint32 uiF;
};

// sizeof(Padded) == 28

```

The configuration above matches the optimal size of 28 bytes, but generally, the optimum cannot be achieved. However, some ISOs rely on small data structure. A typical example is particles systems featuring thousand or more instances, where each particle can be represented by a struct. In this situation, the memory footprint of the unpadded representation may be prohibitive. This is particularly true for 3DRTS running on embedded platforms, where the memory is always limited.

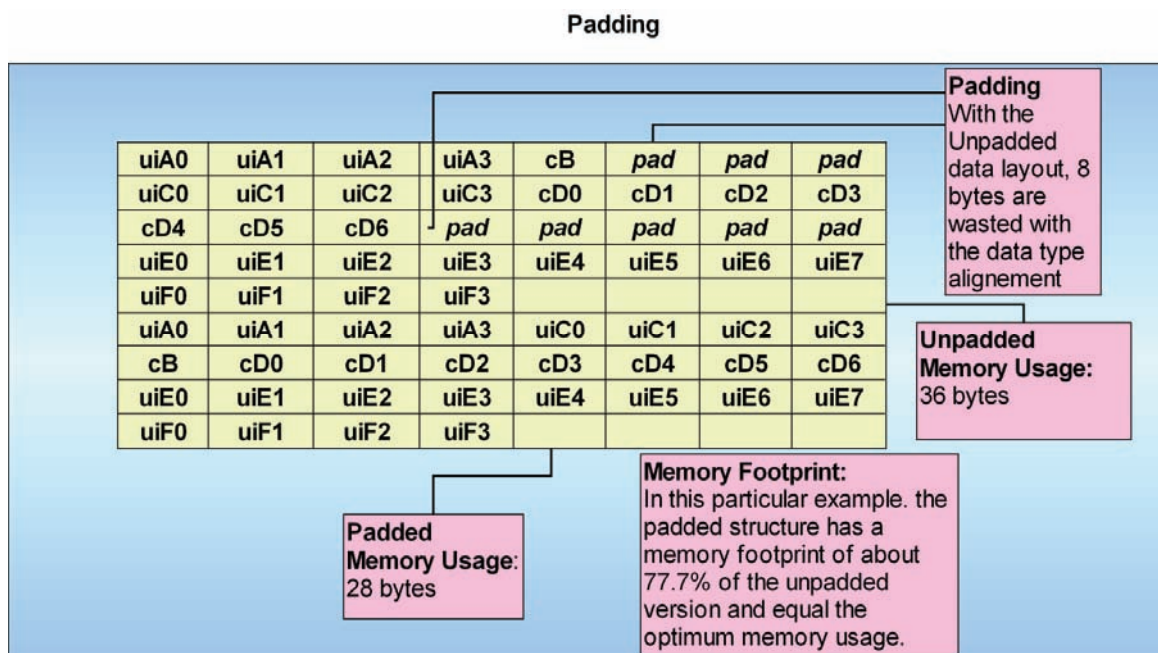


Figure C.2: Memory usage of the same structure based on data alignment constraints.

C.6 Memory Fragmentation

In C++, the operators *new* and *delete* serve to allocate and free objects memory. The platform run-time library provides its own allocator implementation, and is responsible to allocate memory chunks on the heap memory (see Figure C.3). Different heuristics exists such as providing fast allocation routines or to keep memory chunks localized [Ranck2000, Blunden2002]. Unfortunately, none of these heuristics is ideal for all situations. However, recent implementations perform efficiently memory management for general use-case [Berger2001]. Thus, systems architectures developers need to handle memory management for controlling memory fragmentation [Attardi1998, Boer2000, Meyers2002, Glinker2004]. Also handling memory management manually, helps to reduce the OS overheads that need to handle allocation for multiple applications.

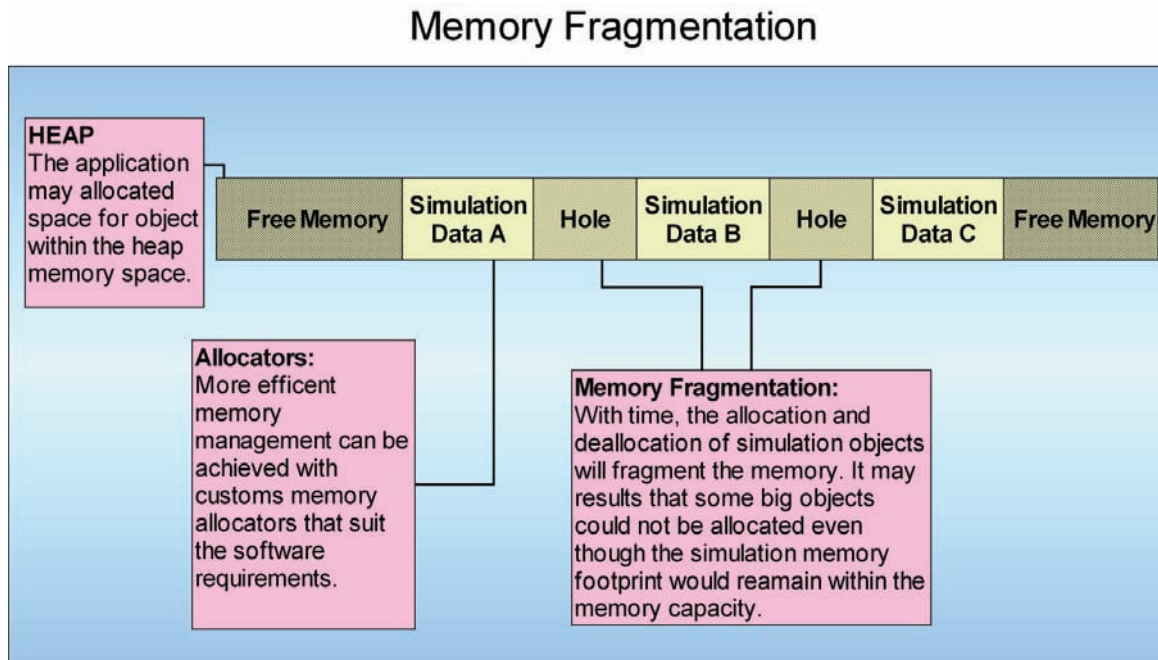


Figure C.3: Memory fragmentation.

This problematic is increase with concurrent system that needs to ensure that minimal data is shared between processes. Also the usage of memory manager overloading the *new* and *delete* operators [Meyers-Op1994] allow to debug memory issues such as memory leaks [Nettle, Dalton2001, Dewhurst2003]. Thus, the creation of different memory containers such as memory pools or buffer for short-time objects is necessary [Hill1992, Gambill2003, Llopis-C++2003, Aue2005]. The Figure C.4 illustrates the different categories of memory allocators, which are separated into two main categories: the heap and stack memory.

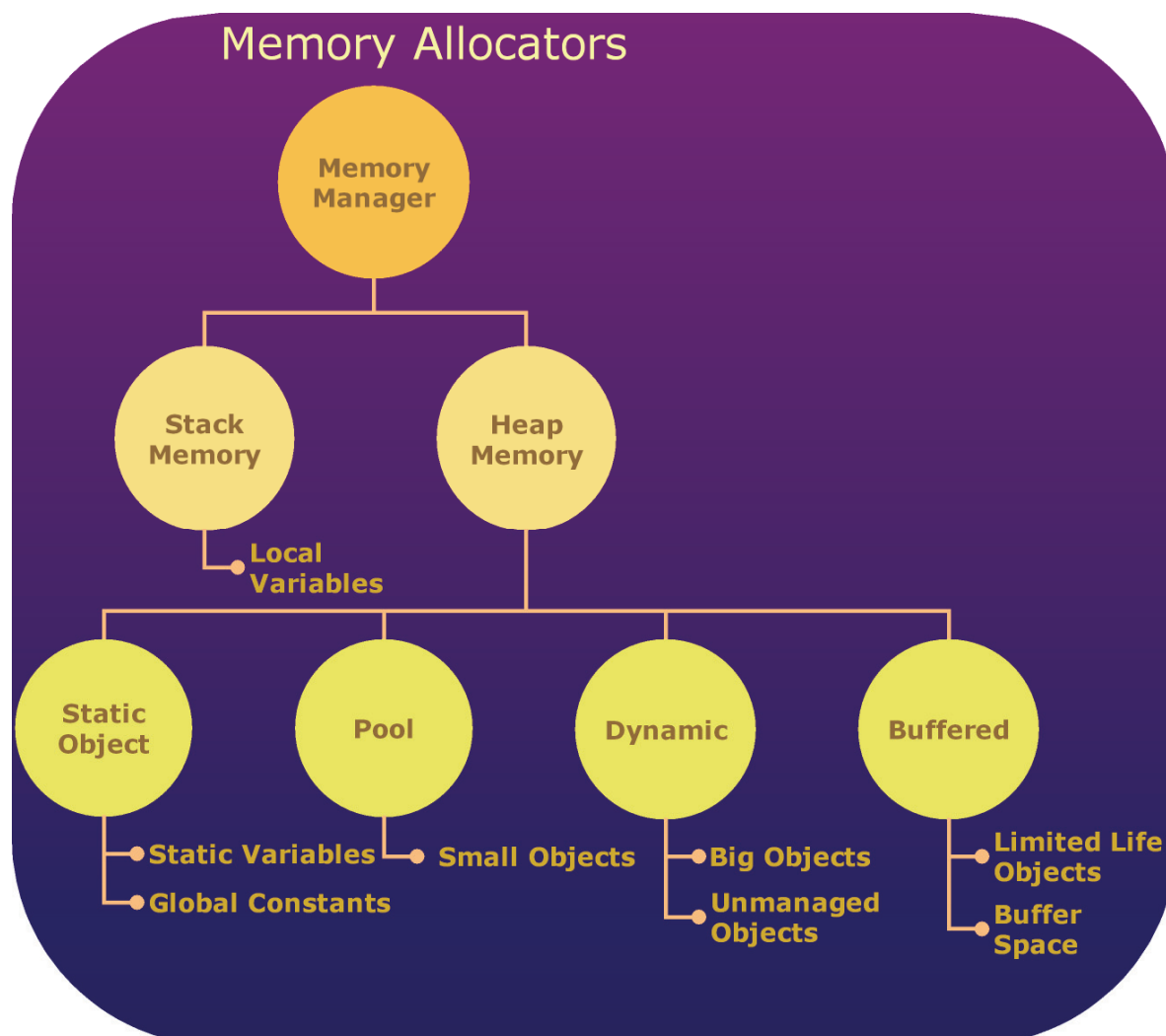


Figure C.4: The different memory allocators.

Appendix D

VHD++ Run-Time GUIs

The VHD++ component framework provides a set of optional vhdGUIWidgets supporting different aspect of developments. They assist both component developers and applications designers along the different development phases including core development, prototyping and testing tasks. The first category of widgets provided practical information about the current systems status. The Figure C.2A show a GUI that list all the supported ISOs, refer in the system as vhdProperties. Based on configuration files and plug-ins different ISOs may be supported. These GUIs help designers to ensure that particular simulation objects will be active and not ignore by the system due to inconsistent configuration files. Figure C.2B shows a GUI allowing for dynamic control of any vhdService component lifecycle (init, run, freeze, terminate). This gives the abilities to freeze some services at run-time for diagnostics and testing purposes. Figure C.2C captures a GUI allowing for inspection of services scheduling, which highlight services running on the same thread. This widget allows investigating the current workflow. This help to distribute tasks among the available hardware threads. The Figure C.2D shows a GUI allowing for inspection of the C++ class hierarchy based on the RTTI information.

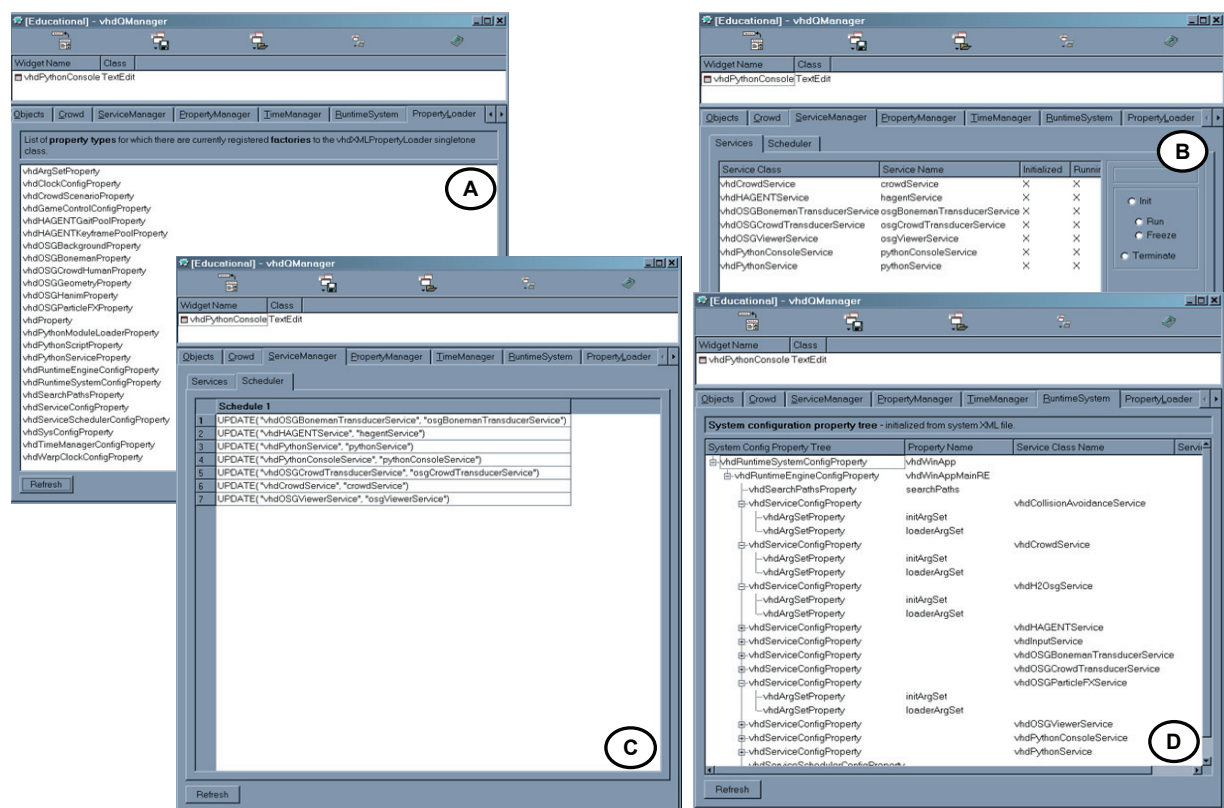


Figure D.1: vhdWidgets providing diagnostic information about the running system.

The next series of vhdWidgets are used at run-time to directly interact with the simulation. The Figure C.3A shows a GUI enabling system inspection and simulation and warp clocks control. This provides the abilities to accelerate or slowing down simulation time. This gives the opportunity to inspect character animations or to analyse if virtual humans' behaviours remain persistent and correct. Figure C.3B is the vhdPythonConsole GUI allowing the dynamic creation and management of python scripts either in a concurrent

or cooperative multitasking environment. Due to the particularities of the Python scripting language, which commute between threads only at the instruction level and by the inner script management, free scripts writers of many issues, related to concurrent programming such as data synchronizations. The scripts also provide behavioural coupling of vhdService components. In particular, it allows for loading, editing, running, pausing or resuming of procedural scripts. In this regard, the system architecture supports rapid prototyping in a data-driven fashion. The Figure C.3C shows a dedicated GUI for convenient modifications of basics parameters such as position and orientation of any movable 3D scene components like objects or virtual characters. Figure C.3D captures a GUI for inspecting the current active ISOs (vhdProperties). It allows as well the ability to dynamically configure, add or remove ISOs from the main aspect-graph defining structural relationship between content side components. Naturally, the system kernel will propagate the information on lower level components. The system flexibility is extending by allowing changing the simulation content at run-time avoiding the need to re-launch the application at every modification.

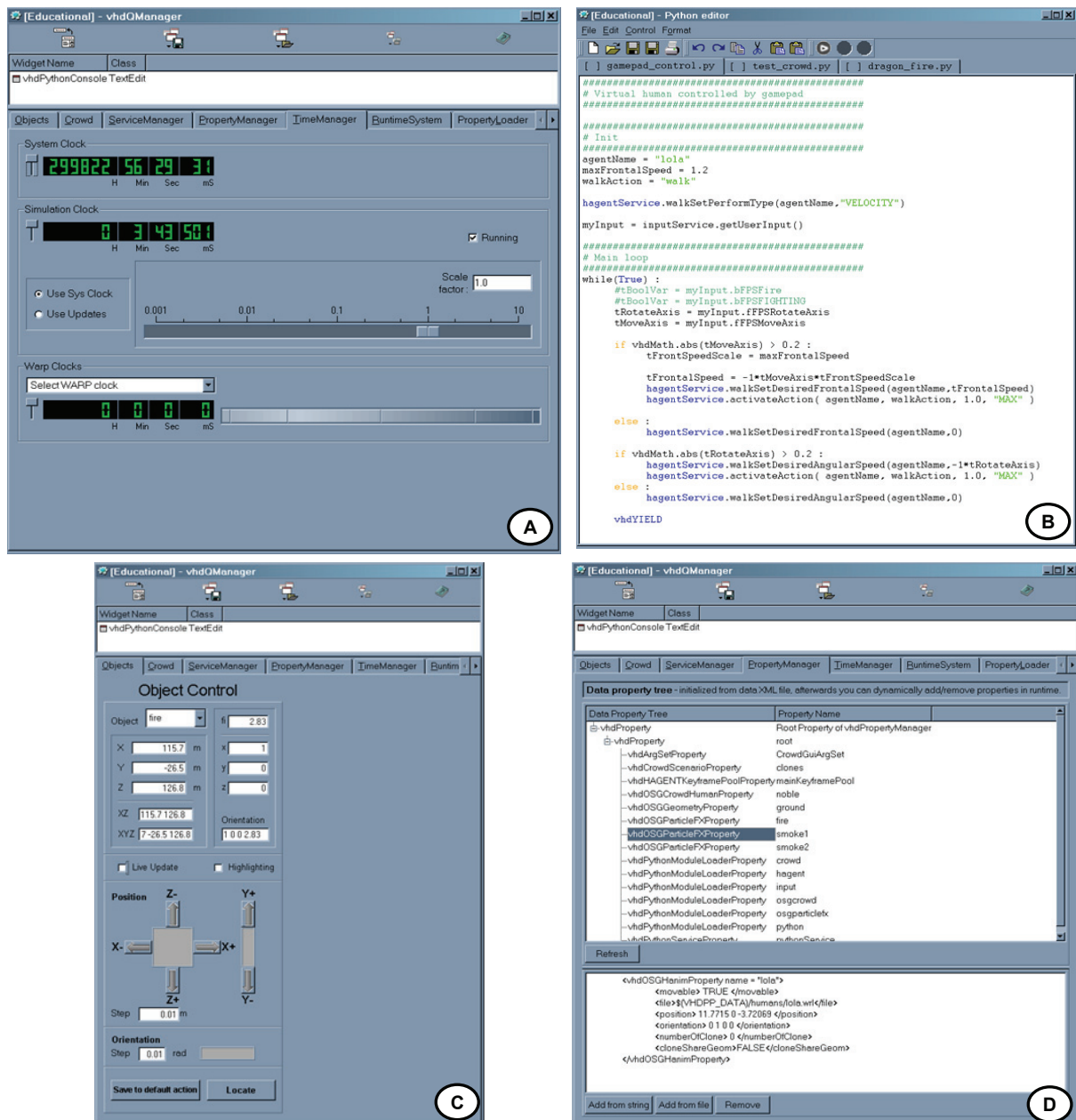


Figure D.2: vhdWidgets for run-time interaction.

The last section of the vhdWidgets provides diagnostics and profiling information at run-time. The Figure C.4A captures a GUI for colour coded monitoring of all diagnostics messages intercept by the vhdDiagManager. It provides set of functionalities that allow component developers to inform the users about potential issues related to the usage of their component. Messages can come under several layers of gravity from simple warning information up to system critical errors similarly than in [Boer2004] or [Duquette2005]. The diagnostic appear with different colour schemes that considerably help developers to analyse situations. In addition, this offers the abilities for end-users to provide backlog files whenever they encounter strange behaviours [McKay2000]. The Figure C.4B highlights one of the two widgets provided by the inner vhdProfiler. This GUI provide graphs of components resources usage over time, helping to discover which modules are CPU and memory intensive but also depict situations where components may have peaks in their resource requirements. This can leads to better tuning. The last widget describe is the one illustrate in the Figure C.4C, which allow to analyse the current usage of every profiled code sections based on the different categories provided by the inner profiler such as hierarchical time or variance. The GUI also reorders the tasks among their resources requirements and offers the ability to store those diagnostics information into MS Excel compatible chart data structures for off-line investigations.

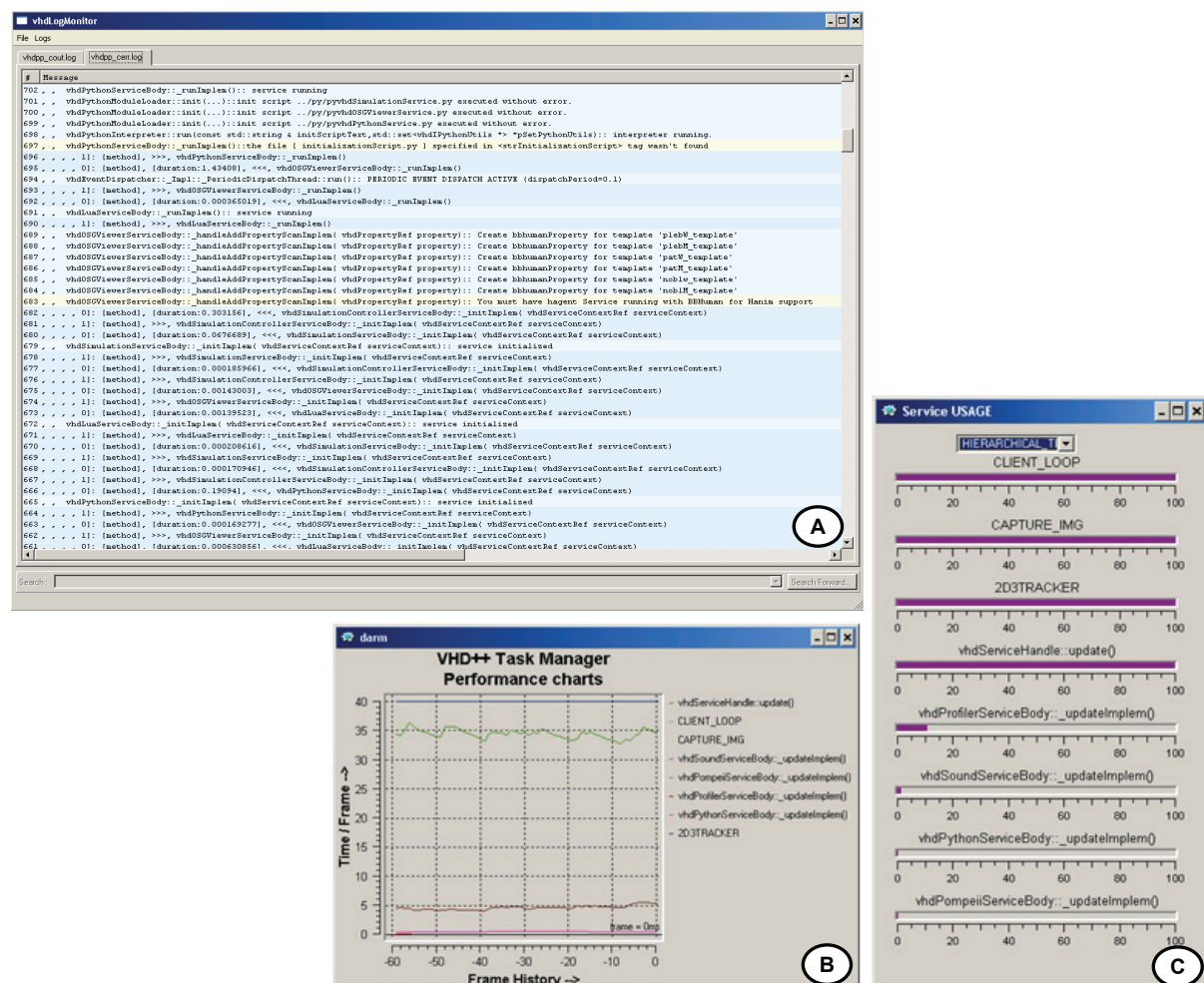


Figure D.3: Diagnostic and profiling widgets.

Appendix E

XML System Configuration File

This appendix illustrates an extract from a full system configuration file, used to specify schedulers, handling component's update as well as specifying which modules need to be loaded by the runtime engine.

Listing vhdSys.xml

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!-- Author: S.Schertenleib -->

<vhdProperty name = "root">

    <!--
    *****
    SERVICES
    *****
    -->
    <vhdServiceSchedulerConfigProperty>
        <schedule>
            <otherServiceUpdates />
            <serviceUpdate className="vhdHAGENTService" name="" />
            <serviceUpdate className="vhdH2OsgService" name="" />
            <serviceUpdate className="vhdSimulationService" name="" />
        </schedule>
        <schedule>
            <serviceUpdate className = "vhdPythonService" name = "" />
            <serviceUpdate className = "vhdSoundService" name = "" />
            <serviceUpdate className="vhdLuaService" name="" />
        </schedule>
        <schedule>
            <serviceUpdate className="vhdOSGViewerService" name="" />
        </schedule>
    </vhdServiceSchedulerConfigProperty>

    <!--
    Here you specify a list services to be loaded by the RuntimeEngine
    -->
    <service className = "vhdPythonService" name = "pythonService" />
    <service className = "vhdOSGViewerService" name = "osgViewerService" />
    <service className = "vhdLuaService" name = "luaService" />
    <service className = "vhdSimulationService" name = "simulationService" />
    <service className = "vhdSoundService" name = "soundService" />
    <service className = "vhdHAGENTService" name = "hagentService" />
    <!--
        <service className = "vhdPathplanningService" name =
            "pathplanningService" />
        <service className = "vhdHObjectService" name = "hobjectService" />
        <service className = "vhdProfilerService" name = "profilerService" />
    -->

    <!--
    Here you specify a list of GUI to be loaed by the RuntimeEngine
    -->
    <vhdGUIConfigProperty name = "vhdPropertyManagerGui"/>
    <vhdGUIConfigProperty name = "vhdPythonConsoleGui"/>
    <vhdGUIConfigProperty name = "vhdObjectControlGui"/>
```



```

<vhdGUIConfigProperty name = "vhdPlayerGui"/>
<vhdGUIConfigProperty name = "vhdERATOControlGui"/>
<!--
  <vhdGUIConfigProperty name = "vhdHanimHumanGUI"/>
  <vhdGUIConfigProperty name = "vhdServiceManagerGui"/>
-->

<!--
  Every Service can support specific option through ArgSetProperty
-->
<vhdServiceConfigProperty className = "vhdOSGViewerService" name = "">
  <vhdArgSetProperty name = "initArgSet">
    <bool name="optimizerOff">TRUE</bool>
    <bool name="lightEnsure">TRUE</bool>
    <vhdColorRGB name="backgroundColor"> 0 0 0 </vhdColorRGB>
    <bool name="viewerOff">FALSE</bool>
    <bool name="stereoFlag">FALSE</bool>
    <string name="stereoMode">ANAGLYPHIC</string>
    <bool name="fullScreenFlag">FALSE</bool>
    <bool name="noBorderFlag" > FALSE</bool>
    <int name="windowPositionX">100</int>
    <int name="windowPositionY">200</int>
    <int name="windowWidth">800</int>
    <int name="windowHeight">600</int>
    <string name="windowName">ERATO Demo</string>
    <float name="idleTime">0.0</float>
    <bool name="HUDFlag">FALSE</bool>
    <bool name="viewerYup">TRUE</bool>
  </vhdArgSetProperty>
</vhdServiceConfigProperty>

</vhdProperty> <!-- ROOT PROPERTY -->

```

Appendix F

XML Data Configuration Files

This appendix illustrates an extract from a full data configuration file, used to configure the initial environment at run-time. The files specify the different properties and their associated resources:

Listing vhdDta.xml

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!-- Author: S.Schertenleib -->
<vhdProperty name = "root">
  <vhdSysConfigProperty>
    <vhdSearchPathsProperty name="dataSearchPaths_001">
      <separator char = "," />
      <path type = "vrm1"> Lifeplus\3Dmodels,</path>
      <path type = "hanim"> Lifeplus\vh,</path>
      <path type = "vrm1">Lifeplus\3Dmodels,</path>
      <path type = "wrk"> Lifeplus\wrk,</path>
      <path type = "FAP"> ../ </path>
    </vhdSearchPathsProperty>
  </vhdSysConfigProperty>
  <!--
  *****
  OSG OBJECTS
  *****
  -->
  <vhdOSGGeometryProperty name = "odeon_windows">
    <movableFlag> TRUE </movableFlag>
    <visibleFlag> TRUE </visibleFlag>
    <file>.\data\erato\scenes\osg\windows\windows.osg</file>
    <position> 0.0 0.0 0.0 </position>
    <orientation> 1 0 0 -1.57 </orientation>
    <numberOfClone> 0 </numberOfClone>
    <cloneShareGeomFlag>TRUE</cloneShareGeomFlag>
  </vhdOSGGeometryProperty>

  <vhdOSGGeometryProperty name = "odeon_ext">
    <movableFlag> TRUE </movableFlag>
    <visibleFlag> TRUE </visibleFlag>
    <file>.\data\erato\scenes\osg\exterior\exterior.ive</file>
    <position> 0.0 0.0 0.0 </position>
    <orientation> 1 0 0 -1.57 </orientation>
    <numberOfClone> 0 </numberOfClone>
    <cloneShareGeomFlag>TRUE</cloneShareGeomFlag>
  </vhdOSGGeometryProperty>
  <!--
  *****
  VR HUMAN
  *****
  -->
  <vhdBBHumanProperty name="nob1M_template">
    <path>data/human/nob1M00</path>
    <tpoFile>data/human/nob1M00/nob1M00.tpo</tpoFile>
    <listAction>idle 4 m_idle10 2.6 m_idle11 1.2 m_idle12 3.0 m_idle13 2.4
      </listAction>
    <listAction>negative 3 protest2 1.0 whistle1 4.0 whist2 4.0</listAction>
    <listAction>positive 2 clap2L 6.0 clap3L 4.2</listAction>
    <listAction>desperate 3 m_cry0 2.2 m_cry1 2.8 m_cry2 2.0</listAction>
    <listAction>laugh 2 m_sitlaugh00 4.0 m_sitlaugh01 2.0</listAction>
    <listAction>listenM 2 m_listen00 2.0 m_listen01 2.0</listAction>
    <listAction>listenT 2 m_listen10 10.0 m_listen11 10.0</listAction>
    <listAction>surprise 1 surprise00</listAction>
```

```

</vhdBBHumanProperty>

<vhdBBHumanProperty name="noblW_template">
  <path>data/human/noblW00</path>
  <tpoFile>data/human/noblW00/noblW00.tpo</tpoFile>
  <listAction>idle 4 idle0 3.0 idle1 3.0 idle3 3.1 idle4 3.3</listAction>
  <listAction>negative 3 whistle1 4.0 whistle1 4.0 bouh1 3.0</listAction>
  <listAction>positive 3 clap1 5.0 clap2 6.0 clap3 4.2</listAction>
</vhdBBHumanProperty>

<vhdBBHumanInstanceProperty name="noblM0">
  <template>noblM_template</template>
  <position>0.0 0.0 0.0</position>
  <orientation>0.0 1.0 0.0 1.52</orientation>
  <scale>0.01</scale>
  <strMaterialName>0</strMaterialName>
</vhdBBHumanInstanceProperty>
<!--
*****
Special Characters (Hi-Fidelity Actors)
*****
-->
<vhdBBHumanInstanceProperty name="senatorM0">
  <template>noblM_template</template>
  <position>0.0 0.0 0.0</position>
  <orientation>0.0 1.0 0.0 1.52</orientation>
  <scale>0.01</scale>
  <strMaterialName>3</strMaterialName>
</vhdBBHumanInstanceProperty>

<vhdBBHumanInstanceProperty name="senatorW0">
  <template>noblW_template</template>
  <position>0.0 0.0 0.0</position>
  <orientation>0.0 1.0 0.0 1.52</orientation>
  <scale>0.01</scale>
  <strMaterialName>3</strMaterialName>
</vhdBBHumanInstanceProperty>
<!--
*****
KEYFRAME ACTIONS TO BE CREATED FOR ALL ACTORS (vhdHAGENTService)
*****
-->
<vhdHAGENTKeyframePoolProperty name = "mainKeyframePool">
  <keyframe name =
"NobleMan">data\erato\animation\nobleM_WalkSaluteSit01.wrk</keyframe>
  <keyframe name =
"NobleWoman">data\erato\animation\nobleW_WalkSaluteSit01.wrk</keyframe>
  </vhdHAGENTKeyframePoolProperty>
<!--
*****
HactionProperty Binding between agent and action
*****
-->
<vhdHactionProperty name="WalkSaluteSit">
  <type>hsa_keyframe</type>
  <agent>senatorM0</agent>
  <vhdArgSetProperty name="hsa_arguments">
    <string name="keyframe">NobleMan</string>
  </vhdArgSetProperty>
</vhdHactionProperty>

<vhdHactionProperty name="WalkSaluteSit">
  <type>hsa_keyframe</type>
  <agent>senatorW0</agent>
  <vhdArgSetProperty name="hsa_arguments">
    <string name="keyframe">NobleWoman</string>
  </vhdArgSetProperty>
</vhdHactionProperty>

```

```

<vhdHactionProperty name="walk">
  <type>hsa_walk</type>
  <agent>NobleMan</agent>
  <agent>NobleWoman</agent>
</vhdHactionProperty>
<!--
*****
PYTHON CONFIG
*****
-->
<vhdPythonServiceProperty name = "pythonService">
  <bPrintRedirection> TRUE </bPrintRedirection>
  <bSynchronizedPythonWithCplusplus> TRUE
  </bSynchronizedPythonWithCplusplus>
  <bSynchronizedPythonWithCplusplusButRunningInParallel> TRUE
  </bSynchronizedPythonWithCplusplusButRunningInParallel>
  <bGarbageCollectorEnabled> FALSE </bGarbageCollectorEnabled>
  <iPythonThreadPriority> MIN_PRIORITY </iPythonThreadPriority>
  <strInitializationScript>initScript.py</strInitializationScript>
</vhdPythonServiceProperty>
<!--
*****
PYTHON CONSOLE CONFIG
*****
-->
<vhdPythonModuleLoaderProperty name = "viewer">
  <strModuleLoader> vhdOSGViewerService </strModuleLoader>
  <strModuleInstance> osgViewerService </strModuleInstance>
  <strPythonScript> ../py/pyvhdOSGViewerService.py </strPythonScript>
</vhdPythonModuleLoaderProperty>

<vhdPythonModuleLoaderProperty name = "simulation">
  <strModuleLoader> vhdSimulationService </strModuleLoader>
  <strModuleInstance> simulationService </strModuleInstance>
  <strPythonScript> ../py/pyvhdSimulationService.py </strPythonScript>
</vhdPythonModuleLoaderProperty>

<vhdPythonScriptProperty name="input">
  <strScriptPath> inputLifeplusGarden.py </strScriptPath>
  <strScriptName> MyScriptIDInput </strScriptName>
  <bYieldScript> TRUE </bYieldScript>
</vhdPythonScriptProperty>
<!--
*****
Lua
*****
-->
<vhdLuaScriptProperty name="motivation">
  <strScriptPath> ./data/lua/motivation.lua </strScriptPath>
</vhdLuaScriptProperty>

<vhdLuaScriptProperty name="fsm">
  <strScriptPath> ./data/lua/fsm.lua </strScriptPath>
</vhdLuaScriptProperty>
<!--
*****
PARAMETERS FOR MAP WIDGET
*****
-->
<vhdMapGuiProperty name = "eratoMapParameters">
  <autoViewpoints>FALSE</autoViewpoints>
  <mapImageFile>data/gui/ERATO/maps/OdeonMap.png</mapImageFile>
  <!-- should be <= 240x200 pixels -->
  <viewerColor>lightblue</viewerColor>
  <viewerSize> 10 </viewerSize>
  <!-- map size (pixels) -->
  <scale> 10 </scale>
  <!-- MapLength (pixels) /RealLength (meters) -->
  <rotation> -1.57 </rotation>

```

```

<!-- (radians) -->
<shift> 0.0 150.0 </shift>
<!-- map size (pixels) -->
<trigger name = "1">
    <color>red</color>
    <position>0.912 14.64 -20.90</position>
    <!-- real world (meters) -->
    <radius>2.0</radius>
    <scriptList>
        <scriptName>data/python/scenarioPart1.py</scriptName>
        <scriptName>data/python/scenarioPart2.py</scriptName>
    </scriptList>
</trigger>
<trigger name = "2">
    <color>red</color>
    <position>-0.2108 6.56 14.16</position>
    <radius>2.0</radius>
    <scriptList>
        <scriptName>data/python/scenarioPart3.py</scriptName>
        <scriptName>data/python/scenarioPart4.py</scriptName>
    </scriptList>
</trigger>
</vhdMapGuiProperty>
<!--
*****
SOUND CONFIG
*****
-->
<vhdSoundServiceConfigProperty name="soundService">
    <bUsing3DSound> FALSE </bUsing3DSound>
    <speakerConfig> DSSPEAKER_STEREO </speakerConfig>
    <bUseEAX3SurroundSound> FALSE </bUseEAX3SurroundSound>
    <fDistanceFactor> 0.1 </fDistanceFactor>
    <fListenerRolloff> 0.1 </fListenerRolloff>
    <fDopplerFactor> 1.0 </fDopplerFactor>
    <bScaleDecayTime> TRUE </bScaleDecayTime>
    <bScaleReflections> TRUE </bScaleReflections>
    <bScaleReflectionsDelay> TRUE </bScaleReflectionsDelay>
    <bScaleReverb> TRUE </bScaleReverb>
    <bScaleReverbDelay> TRUE </bScaleReverbDelay>
    <lVolume> 0 </lVolume>
</vhdSoundServiceConfigProperty>
<!--
*****
BACKGROUND SOUNDS
*****
-->
<vhdSoundMediaPlayerProperty name="Part1.0_intro">
    <filename>data/sound/Part1.0_intro.wav</filename>
    <bLoopMode> FALSE </bLoopMode>
    <bAudioOnly> TRUE </bAudioOnly>
    <lVolume> 0 </lVolume>
    <lBalance> 0 </lBalance>
</vhdSoundMediaPlayerProperty>
<!--
*****
Input Service Config
*****
-->
<vhdGameControlConfigProperty name = "gait_demo">
    <showConfigGui>FALSE</showConfigGui>
    <debugDump>FALSE</debugDump>
    <controlledAgent>lola</controlledAgent>
    <agentControlEnabled>FALSE</agentControlEnabled>
    <maxFrontalSpeed>1.2</maxFrontalSpeed>
</vhdGameControlConfigProperty>
</vhdProperty>

```


Appendix G

Python Scripts

The next listings illustrate sections of microthread scripts controlling simulation events at run-time.

Listing Managing Input Devices within a Microthread:

```
event = UserInputPtr(inputService.getUserInput())
while (True):
    if event.bFPSFire == True:
        #do event management
        pythonService.executeYieldScript("scene_v10.py")
    elif event.fFPSMoveAxis != 0:
        #do event ...
    #... other input
    vhdYIELD #wait until next update
```

Listing Walking Animation Script using HWALK:

```
celer="Celer"
tDest = vhdVector3(-30.0, 0.0, -10.0)
hagentService.walkSetPerformType( celer , "POSITION")
hagentService.walkSetPositionReachedTolerance( celer , 0.3)
hagentService.walkSetDesiredFrontalSpeed( celer , 0.6 )
hagentService.walkSetHaltingOption( celer , False)

print "walk to"
hagentService.walkSetDesiredPosition(celer , tDest)
hagentService.walkSetDesiredPosition(celer , tDest)
hagentService.activateAction(celer , "walk", 1.0, "MAX")
```

Listing for controlling Particle System:

```
rain = osgParticleFXService.getParticleSystem("rain")
rain.setParticleAlphaInterpolator3(0.5,0.05)
rain.setParticleLifeTime(5)
rain.setParticleMass(0.1)
rain.setParticleRadius(0.01)
rain.setParticleSizeInterpolator(0.1,0.2)
rain.setShooterThetaRange(0,1)
while (True):
    frame = frame + 1
    if frame > 300:
        rain.setCounterRange(0,0)
    elif frame > 600:
        rain.setCounterRange(30,30)
    vhdYIELD
```

Listing for controlling a virtual human using a gamepad:

```
inputService.setControlledAgentName("Vescla")
# Action names are hardcoded in vhdInputService as they are reassignable with
# DirectInput Dialog (originally they come from FPS mapping)
#we associate a python script with an action
inputService.setPythonScriptFile("Fire", "./python/voice_specula_hi.py")
inputService.setPythonScriptFile("SelectWeapon", "./python/voice_ascla_hi.py")
inputService.setPythonScriptFile("UseItem", "")
inputService.setPythonScriptFile("SelectItem", "./python/control_specula.py")
inputService.setPythonScriptFile("Crouch", "./python/control_ascla.py")
inputService.setPythonScriptFile("Jump", "")
```

Listing for rotating a skybox:

```
vhdQuaternion quatRot = vhdQuaternion(0.0,0.0,1.0,0.0)
while (True):
    tAngle = quatRot.getAngle()
    tAngle = tAngle + 0.0005
```

```
tRot.setAngle(tAngle)
hagentService.setRotation("skybox", quatRot)
```

Listing HUD for the manipulation 2D Widgets

```
color = vhdVector4(0.0,1.0,0.0,1.0)
osgViewerService.setHUDTextColor(color)
osgViewerService.setHUDTextSize(40)
osgViewerService.setHUDTextFont("Fonts/georgia.ttf")
pos = vhdVector2(100,100)
osgViewerService.setHUDTextPosition(pos)
osgViewerService.setHUDText("Hello")
```

Listing for defining and SE environment for PathPlanning Request:

```
#define the domain limits
v=[-2.8,-6.9,11.5,-6.9,11.5,6.9,-2.8,6.9]
pathPlanningService.createMap(v)
pathPlanningService.getObstacle("obstacles.xml")
for i in range(len(obstacles))
    pathPlanningService.insertObstacle(obstacles[i])
```

Listing Controlling Smart Object:

```
hobjectAppleA1 = libhobject.ObjectoidPtr(
    hobjectService.getHObjectByName("appleA1obj"))
hobjectAppleA2 = libhobject.ObjectoidPtr(
    hobjectService.getHObjectByName("appleA2obj"))

anim = hobjectFigaV2.getAnimatorByName("appleA1_Anim0-184")
anim.start()
anim = hobjectBasketL1.getAnimatorByName("appleA2_Anim185-323")
anim.start()
```

Listing to Move and object and a virtual human using SmartObjects:

```
#connect virtual human and object
eH = PythonEventHandler(g_pyMicroThreadNameList(g_pyCurrentMicroThreadInRun])
oS = hobjectService
human = Human("myHuman")
box = ObjectoidPtr(oS.geHObjectByName("ObjectZbox"))

#move human toward box2
human.walk(asAttributeSet(box.getNodeByName("ZboxPos2")), Human.Stop)
eH.subscribe(WalkReachedEventClass(human))
eh.pauseScript()
vhdYIELD
ev = eH.checkEvents()
print ev

#move human toward box
human.walk(asAttributeSet(box.getNodeByName("ZboxPos")), Human.Stop)
eH.subscribe(WalkReachedEventClass(human))
eH.pauseScript()
vhdYIELD
ev = eH.checkEvents()
for i in range(10):
    vhdYIELD

#set up Inverse Kinematics Goal
rHas = asAttributeSet(box.getNodeByName("ZboxHand"))
human.setIKGoal(human.RightArm,
rHAS.lookupUpdated("transform").getMatrix())
#wait until no more pending event
while not eH.eventsPending():
    human.UpdateObjectFromRightHand(box, rHO)
vhdYIELD
```

Listing to load H-ANIM compliant WRK keyframe animation:

```
hagentService.kfLoad("mann","data\Animations\WRK\applause00.wrk")
hagentService.kfCreate("Hactor_Gabby", "cool", "mann")
hagentService.activateAction("Hactor_Gabby", "cool", 1.0, "MAX")
```

Listing Excerpt from a LIFEPLUS scenario:

```

import libhobject
#####
#          SCENES 1, 2, 3, 4, 5
#####

color = vhdVector4(0.0, 0.0, 1.0,1.0)
osgViewerService.setHUDTextColor(color )
pos = vhdVector2(50,125)
osgViewerService.setHUDTextPosition(pos )
osgViewerService.setHUDTextSize(22 )
osgViewerService.setHUDTextFont("fonts/Arialbd.ttf" )
osgViewerService.setHUDText("BDGP and BDSeb are proud to present you this
beautiful ANSI-compliant AR/VR Demo:\nLIFEPLUS (taverna)!")

# Script Global Variable
ascla="Ascla"
vetutius="Vetutius"
lucius="Lucius"
celer="Celer"
specula="Specula"
vecPos=vhdVector3()
vecRot=vhdQuaternion()
vecAxis=vhdVector3()
hobjectAppleA1                                     =
libhobject.ObjectoidPtr(hobjectService.getHObjectByName("appleA1obj"))
hobjectAppleA2                                     =
libhobject.ObjectoidPtr(hobjectService.getHObjectByName("appleA2obj"))
#####
# Init of object position
#####
anim = hobjectAppleA1.getAnimatorByName("appleA1_AnimInit")
anim.start()
anim = hobjectAppleA2.getAnimatorByName("appleA2_AnimInit")
anim.start()
#####
# Init characters position and orientation
#####
#celer
hagentService.stopAllActions(celer)
pos = vhdVector3(1.91, 0.0, -2.64)
hagentService.setPosition(celer,pos)
rot = vhdQuaternion(-1.059, 0.0, 1.0, 0.0)
hagentService.setRotation(celer,rot)

#Vetutius
hagentService.stopAllActions(vetutius)
pos = vhdVector3(0.463, 0.013, 0.647)
hagentService.setPosition(vetutius,pos)
rot = vhdQuaternion(-1.57, 0.0, 1.0, 0.0)
hagentService.setRotation(vetutius,rot)

#ascla
hagentService.stopAllActions(ascla)
pos = vhdVector3(1.733,0.0,-0.5)
hagentService.setPosition(ascla,pos)
rot = vhdQuaternion(3.33996,0.0, 1.0, 0.0)
hagentService.setRotation(ascla,rot)

#Specula
hagentService.stopAllActions(specula)
pos = vhdVector3(-3.37, 0.0, -2.72)
hagentService.setPosition(specula,pos)
rot = vhdQuaternion(3.14, 0.0, 1.0, 0.0)
hagentService.setRotation(specula,rot)

#Lucius
hagentService.stopAllActions(lucius)

```

```

pos = vhdVector3(-1.26, -0.032, -2.549)
hagentService.setPosition(lucius,pos)
rot = vhdQuaternion(0.0, 0.0, 1.0, 0.0)
hagentService.setRotation(lucius,rot)

#####
# SCENE 1
#####
#Ascla and Vetutius are preparing food
hagentService.kfSetGlobalTranslationMode(ascla,'S100_No_Ascla',"KEYFRAME_RELATIVE_MODE")
hagentService.kfSetGlobalRotationMode(ascla,'S100_No_Ascla',"KEYFRAME_ABSOLUTE_MODE")
hagentService.activateAction(ascla,'S100_No_Ascla',0.0)
anim = hobjectAppleA1.getAnimatorByName("appleA1_Anim0-102")
anim.start()
anim = hobjectAppleA2.getAnimatorByName("appleA2_Anim0-184")
anim.start()
hagentService.kfSetGlobalTranslationMode(vetutius,'S100_No_Vetutius',"KEYFRAME_RELATIVE_MODE")
hagentService.kfSetGlobalRotationMode(vetutius,'S100_No_Vetutius',"KEYFRAME_ABSOLUTE_MODE")
hagentService.activateAction(vetutius,'S100_No_Vetutius',0.0)
bRun=True
while (bRun==True):
    if (hagentService.kfHasReachedEndOfSequence(vetutius,'S100_No_Vetutius')):
        hagentService.stopAction(ascla,'S100_No_Ascla')
        hagentService.stopAction(vetutius,'S100_No_Vetutius')
        bRun=False
    vhdYIELD

#Africanus talks behind
hagentService.kfSetGlobalTranslationMode(vetutius,'S101_F_Vetutius',"KEYFRAME_RELATIVE_MODE")
hagentService.kfSetGlobalRotationMode(vetutius,'S101_F_Vetutius',"KEYFRAME_ABSOLUTE_MODE")
hagentService.activateAction(vetutius,'S101_F_Vetutius',0.0)
anim = hobjectFigaV2.getAnimatorByName("figaV2_Anim0-184")
anim.start()
anim = hobjectBasketL1.getAnimatorByName("basketL1_Anim185-323")
anim.start()
africanus1="root.Africanus1"
osgViewerService.setHUDText("Africanus say: Vetutius, bring me one more jug of Falernum jug,\n and make sure it is the good one!")
soundService.sndmpPlayMedia(africanus1)
hagentService.kfSetGlobalTranslationMode(ascla,'S101_F_Ascla',"KEYFRAME_RELATIVE_MODE")
hagentService.kfSetGlobalRotationMode(ascla,'S101_F_Ascla',"KEYFRAME_ABSOLUTE_MODE")
hagentService.activateAction(ascla,'S101_F_Ascla',0.0)
anim = hobjectSpongeA1.getAnimatorByName("spongeA1_Anim185-323")
anim.start()
hagentService.kfSetGlobalTranslationMode(lucius,'S101_F_Lucius',"KEYFRAME_RELATIVE_MODE")
hagentService.kfSetGlobalRotationMode(lucius,'S101_F_Lucius',"KEYFRAME_ABSOLUTE_MODE")
hagentService.activateAction(lucius,'S101_F_Lucius',0.0)
bRun=True
while (bRun==True):
    if (hagentService.kfHasReachedEndOfSequence(lucius,'S101_F_Lucius')) :
        hagentService.stopAction(vetutius,'S101_F_Vetutius')
        hagentService.stopAction(ascla,'S101_F_Ascla')
        hagentService.stopAction(lucius,'S101_F_Lucius')
        soundService.sndmpStopMedia(africanus1)
        bRun=False
    vhdYIELD
#####
# SCENE 2 ...
#####

```

Appendix H

Lua Scripts

The following listing describes sections of a Lua script, which serve to define virtual humans' behavioral states using Lua metatable and HFSM:

Listing Virtual Human Behaviors:

```
-----
-- State FSM to be used with a iCharacter
-----luaPrint
= vhd.LuaPrint()

FOLLOW_AGENT_EPSILON      = 1.5 -- distance between two agent when one
follow_another
ACTION_IDLE                = "idle"
ACTION_POSITIVE            = "positive"
ACTION_NEGATIVE            = "negative"
ACTION_SURPRISE            = "surprise"
ACTION_FEAR                = "fear"
ACTION_ANGRY               = "angry"
ACTION_DESESPERATE        = "desesperate"
ACTION_LAUGH               = "laugh"
ACTION_UNISON              = "unison"
ACTION_LISTENM             = "listenM"
ACTION_LISTENT             = "listenT"

AGENT_EPSILON              = 1.5
ACTION_PROVIDE_HELP       = "ACTION_PROVIDE_HELP"
GOAL_EPSILON               = "GOAL_EPSILON"
PATH_EPSILON               = 1.0

AGENT_DISTANCE_PERCEPTION = 1.5
ACTION_BLENDING_COMPLETION = 0.5 -- need to complete this value before
blending with potential next anim

-----
create the IDLE Behavior
-----

-----
create the IdleStart state
-----
State_IdleStart = {}

State_IdleStart["Name"] = "State_IdleStart"

State_IdleStart["Enter"] = function(character)
    character:attachTask("SimpleTask",0)
    -- nop
end

State_IdleStart["Execute"] = function(character)
    character:getFSM():changeState(State_IdleWait)
end

State_IdleStart["Exit"] = function(character)
    -- nop
end

-----
create the IdleWait state
```



```

-----
State_IdleWait = {}

State_IdleWait["Name"] = "State_IdleWait"

State_IdleWait["Enter"] = function(character)
    character:attachTask("SimpleTask",0)
    -- nop
end

State_IdleWait["Execute"] = function(character)
    character:doAction(ACTION_IDLE, false, 1.0)
    character:getFSM():changeState(State_IdleActing)
end

State_IdleWait["Exit"] = function(character)
    -- nop
end

-----
create the IdleActing state
-----
State_IdleActing = {}

State_IdleActing["Name"] = "State_IdleActing"

State_IdleActing["Enter"] = function(character)
    character:attachTask("SimpleTask",0)
    -- nop
end

State_IdleActing["Execute"] = function(character)
    if character:isActionNearCompletion(
        ACTION_IDLE, ACTION_BLENDING_COMPLETION) then
        character:doAction(ACTION_IDLE, false, 0.5 + math.random())
    end
end

State_IdleActing["Exit"] = function(character)
    -- nop
end

-----
create the RANDOM EMOTION Behavior
-----
State_RandomStart = {}

State_RandomStart["Name"] = "State_RandomStart"

State_RandomStart["Enter"] = function(character)
    character:attachTask("SimpleTask",0)
end

State_RandomStart["Execute"] = function(character)
    -- random between different emotion
    local nbEmotion = 3
    local res = math.random() * nbEmotion
    if res < 1 then
        character:getFSM():changeState(State_IdleStart)
    elseif res < 2 then
        character:getFSM():changeState(State_PositiveStart)
    elseif res < 2 then
        character:getFSM():changeState(State_PositiveStart)
    elseif res < 3 then
        character:getFSM():changeState(State_NegativeStart)
    end
end

State_RandomStart["Exit"] = function(character)

```

```

-- nop
end

-----

-- create Look and Search Behavior
-----

-----
create the Senses state
-----
State_Senses = {}

State_Senses["Name"] = "State_Senses"

State_Senses["Enter"] = function(character)
    character:attachTask("SimpleTask",0)
    -- nop
end

State_Senses["Execute"] = function(character)
    -- perform the sensing behavior, e.g look, hear, ...
    character:performSensing()
    character:getFSM():changeState(State_Senses)
end

State_Senses["Exit"] = function(character)
    -- nop
end

-----

create the Pascal Walk Behavior
-----

-----
create the IdleStart state
-----
State_PascalWalk = {}

State_PascalWalk["Name"] = "State_PascalWalk"

State_PascalWalk["Enter"] = function(character)
    character:attachTask("SimpleTask",0)
    -- nop
end

State_PascalWalk["Execute"] = function(character)
    character:doAction(ACTION_IDLE, true, 1.0)
end

State_PascalWalk["Exit"] = function(character)
    -- nop
end

-----

create the Follow Path Behavior
-----

-----
create the State_PathBehaviorActivateGoal state
-----
State_PathBehaviorActivateGoal = {}

State_PathBehaviorActivateGoal["Name"] = "State_PathBehaviorActivateGoal"

State_PathBehaviorActivateGoal["Enter"] = function(character)
    character:attachTask("SimpleTask",0)
    -- nop
end

```

```

State_PathBehaviorActivateGoal["Execute"] = function(character)
    character:stopWalk()
    character:walkTo(character:getCurrentPosition(), true)
    character:activateWalk()

    character:getPath():setNextWaypoint()
    if character:getPath():finished() then
        --done
        character:getFSM():changeState(State_PathBehaviorDone)
    else
        character:getFSM():changeState(State_PathBehaviorSelectGoal)
    end
end

State_PathBehaviorActivateGoal["Exit"] = function(character)
    -- nop
end

-----
create the State_PathBehaviorSelectGoal state
-----
State_PathBehaviorSelectGoal = {}

State_PathBehaviorSelectGoal["Name"] = "State_PathBehaviorSelectGoal"

State_PathBehaviorSelectGoal["Enter"] = function(character)
    character:attachTask("SimpleTask",0)
    -- nop
end

State_PathBehaviorSelectGoal["Execute"] = function(character)
    if character:getPath():finished() then
        --done
        character:getFSM():changeState(State_PathBehaviorDone)
    else
        local thalt = false
        character:walkTo(character:getPath():currentWaypoint(), thalt)
        character:getFSM():changeState(State_PathBehaviorGoGoal)
    end
end

State_PathBehaviorSelectGoal["Exit"] = function(character)
    -- nop
end

-----
create the State_PathBehaviorGoGoal state
-----
State_PathBehaviorGoGoal = {}

State_PathBehaviorGoGoal["Name"] = "State_PathBehaviorGoGoal"

State_PathBehaviorGoGoal["Enter"] = function(character)
    character:attachTask("SimpleTask",0)
    -- nop
end

State_PathBehaviorGoGoal["Execute"] = function(character)
    if
character:reachedGoalPosition(character:getPath():currentWaypoint(),PATH_EPSILO
N) then
        character:getPath():setNextWaypoint()
        if character:getPath():finished() then
            --done
            character:getFSM():changeState(State_PathBehaviorDone)
        else
            -- go to next point
            character:getFSM():changeState(State_PathBehaviorSelectGoal)

```

```

        end
    end
end

State_PathBehaviorGoGoal["Exit"] = function(character)
    -- nop
end

-----
create the State_PathBehaviorDone state
-----
State_PathBehaviorDone = {}

State_PathBehaviorDone["Name"] = "State_PathBehaviorDone"

State_PathBehaviorDone["Enter"] = function(character)
    character:attachTask("SimpleTask",0)
    -- nop
    character:stopWalk()
end

State_PathBehaviorDone["Execute"] = function(character)
    -- we return from this state and check if some subgoal have to be done
    character:getFSM():popSubGoal()
end

State_PathBehaviorDone["Exit"] = function(character)
    -- nop
end

-----
Place location
-----
Place = {}
Place["Mine"] = vhd.vhdVector3(23,0,0)
Place["House"] = vhd.vhdVector3(-123,0,10)
Place["Lake"] = vhd.vhdVector3(106,0,0)
Place["PP"] = vhd.vhdVector3(-60,0,8)
Place["Circus"] = vhd.vhdVector3(227.0, 0.0, -8.0)
Place["Station"] = vhd.vhdVector3(-7.5, 0.0, -103.2)
Place["Hotel"] = vhd.vhdVector3(127.5, 0.0, 83.2)
Place["Parc"] = vhd.vhdVector3(-1.5, 0.0, -57.5)
Place["Building"] = vhd.vhdVector3(50.0, 0.0, -35.0)

function goToPlace(character, place)
    -- ask the system to retrieve a path
    character:getPathplanning(place)
end

function findLocation(character, place)
    -- find the a random pt near the place
    return character:getTargetWithinArea(place)
end

```


Appendix I

Build System

Our architecture rely on [Scons] as a build system. We have extended the basic capabilities of SCons for handling our specific build options. The benefit of our home custom build system is twofold. It allows overcoming the limitation of make:

- No Built-in dependency tools
- Dependence based on time-stamps:
 - Clock skew in networked development system
 - Fooled by restoring old source.
- Make does not know that changing compiler flags mean different objects (debug flags / various optimization flags, pre-processor definitions).
- Harder to create multi-directory projects with all libraries and dependencies
- Unreliable need to do many 'make clean' to be sure
- Multiple version of make exist

The major advantages of SCons include:

- Support C/C++, Lex, Yacc, tar, zip, rcs, sccs, cvs, bitkeeper, perforce, precompiled headers, MS resource files, generate .vcproj, MD5 signature, automatic dependencies checking,...
- Cross-platform, help to install upon many platforms: .tar.gz, .zip .exe, .rpm
- Open Source: MIT License

SCons require a Sconstruct file which handle the reliability of the system, by ensuring that all the dependencies are re-builds whenever it is required. Then every library contains a SConscript that specify the objects files as well as the libraries that required to be linked against. In our system, we use a cfg folder, which keeps track of different configuration files:

```
/Sconstruct
/vhdLibName/
    *.cpp
    *.h
    *.i
    *.idl
    *.ui
    SConscript

/cfg/
    config.[platform]
    libname.[platform]
    modules.lst.[platform]
```

Listing config.[platform]:

```
[build]
configuration_name = release
install_root = ../VHDPP
build_root = ../VHDPP_BUILD
modules_list = ./cfg/modules.lst.win32
libnames = ./cfg/libname_release.win32
tmp_path = ../VHDPP_BUILD_TMP

optional_components = VHD_PYTHON VHD_HWALK VHD_LUA USE_VHD_PROFILER QMANAGER
VHD_BBHUMAN THREADS_UNSAFE VHDBB_HAGENT BB_MULTI_THREAD

[compiler]
CC = cl
CXX = cl
```

```

SWIG = swig.exe
SWIGFLAGS=-c++ -python -I./vhdKernel
MSVS_VERSION = 8.0
CFLAGS = -MD -nologo -EHsc -GR -D_CPPRTTI -O2 -Ob2 -Zc:wchar_t -Gd -Gy -openmp -GL
-Wp64 -arch:SSE2 -Ox -fp:fast
CXXFLAGS = -MD -nologo -EHsc -GR -D_CPPRTTI -O2 -Ob2 -Zc:wchar_t -Gd -Gy -openmp -
GL -Wp64 -arch:SSE2 -Ox -fp:fast
CPPFLAGS_GLOBAL = -DVHD_BOOST -DWIN32 -DNDEBUG

component_root = ./components
INCLUDE_PATH = ../VHDPP_TEST/include %(component_root)s/include
LIB_PATH = ../VHDPP_TEST/lib %(component_root)s/lib/
LDFLAGS = /FIXED:NO /NODEFAULTLIB:libc /NODEFAULTLIB:libcmt /NODEFAULTLIB:msvcrt
/NODEFAULTLIB:libcd /NODEFAULTLIB:libcmt /INCREMENTAL /MANIFEST:NO

```

Listing libname.[platform]:

```

#+-----+
#| key      library name |
#+-----+

dinput      dinput8
OpenAL       OpenAL32
opengl       opengl32
OpenThreads  OpenThreadsWin32
osg          osg
python       python24
qt           qt-mt330

```

Listing modules.lst.[platform]:

```

LibvhdMath
vhaABTTree
vhdcegui
vhdEl
vhdHAGENTService
vhdInputEngine
vhdInputService
vhdKernel
vhdLuaInterpreter
vhdLuaService
vhdLZOEngine
vhdOgreService
vhdOgreGUI
vhdOSGViewerService
vhdPathPlanningService
vhdProfiler
vhdProfilerService
vhdPythonInterpreter
vhdPythonConsoleGUI
vhdPythonInterpreter
vhdPythonModuleLoader
vhdPythonService
vhdQManager
#....
vhdResourceManager
vhdSimulationService
vhdSimulationControllerService
vhdTaskScheduler
#...

```

Listing SConscript (here for vhdPythonService):

```

#####
# Env
#####
Import('env')
env.Import('*')

```

```
#####
# Source or special settings for this particular lib, YOU CAN EDIT THIS PART
#####
#VARIABLE THAT YOU CAN USE ARE
#env, source, include

libsMapped = []
source=Split("""
vhdPythonServiceBodyCORBA.idl
_vhdPythonUtils.cpp
    _vhdPythonUtilsPropertyManager.cpp
    _vhdPythonModuleLoaderProperty.cpp
    vhdPythonModuleLoaderPropertyFactory.cpp
vhdPythonScriptProperty.cpp
    vhdPythonScriptPropertyFactory.cpp
vhdPythonServiceBody.cpp
vhdPythonServiceLoader.cpp
vhdPythonServiceProperty.cpp
vhdPythonServicePropertyFactory.cpp
pyvhdPythonService.i
vhdDllServiceInit.cpp
""")

include=Split('''
_vhdPythonUtils.h
    _vhdPythonUtilsPropertyManager.h
    vhdPythonModuleLoaderProperty.h
    vhdPythonModuleLoaderPropertyFactory.h
    vhdPythonScriptProperty.h
    vhdPythonScriptPropertyFactory.h
    vhdPythonServiceBody.h
    vhdPythonServiceLoader.h
    vhdPythonServiceProperty.h
    vhdPythonServicePropertyFactory.h
    vhdIPythonService.h
    vhdPythonServiceBody_i.h
''')

libs = Split('''
    libvhdMath
    vhdKernel
    vhdPythonInterpreter
    vhdPythonModuleLoader
''')

if 'VHD_LUA' in optional_components:
    libs=libs+Split('''
        lua
        luabind
        vhdLuaService
    ''')

if 'USE_VHD_PROFILER' in optional_components:
    libs=libs+['vhdProfiler']

if 'VHD_CORBA' in optional_components:
    libs += ['libvhdOmniORB']
    libsMapped += ['omniORB','omnithread','msvcstub','Ws2_32','Advapi32']

libsMapped+=Split('''
    OpentThreads
    xerces
''')

if env['PLATFORM']=='win32' and env['typeProject'] == 'dynamicLib':
    env.Append(CPPDEFINES = ['VHD_PYTHON_LIBRARY'])
```

```

# explicit dependencies for SWIG
for h in include:
    Depends('pyvhdPythonService_wrap.cpp', h)

#####
#####
# What has to be installed for this module, YOU CAN EDIT THIS PART
#####
#####

installIncludeList = include
installPyList = ['pyvhdPythonService.py']
installIDLList = ['vhdPythonServiceBodyCORBA.idl']
installBinList = []

#####
#####
# Exportation of your current settings
#####
#####

env.Export('source')
env.Export('include')
env.Export('libs')
env.Export('libsMapped')
env.Export('installIncludeList')
env.Export('installBinList')
env.Export('installPyList')
env.Export('installIDLList')

Export('env')

```

Listing : Exerpt from main Sconstruct :

```

#####
Class holding all global configuration loaded from file
#####
s GlobalConfig:
    """ Global config for the build
    """
    platform      = ""          # which platform we are building on
    source_root   = ""          # where is the original source code
    config        = None        # parsed config file
    cfg_root      = ""          # where is the configuration
    modules_list  = []          # list of modules to build for this platform
    libmap        = {}          # dictionary of DLL names for this platform and build

    def __init__(self, configFile):
        """ Constructor, initialize the build
        """
    def load_modules(self, fname):
        """ Load list of modules to be built for this release and
            platform
        """
    def load_libmap(self, fname):
        """ Loads mapping table for DLL names for this release and platform
        """
    def detect_platform(self):
        """ Detect the platform we are running on
            Supported platforms :
            - i586 (Linux on Intel)
            - amd64 (Linux on Opteron)
            - win32 (Windows on Intel)
        """
#####
option supported by our custom build system

```

```
#####opt
ion are set in the custom.py
opts = Options('custom.py')
opts.Add(BoolOption('dump', 'Set to True to dump the global environment', 'no'))
opts.Add(BoolOption('bMakeMSVCSolution', 'Set to True to build the Visual Studio
Solution', 'yes'))
opts.Add(EnumOption('typeProject', 'Choose the type of libraries built',
'dynamicLib',
                    allowed_values = ('staticLib', 'dynamicLib', 'program')))
opts.Add(BoolOption('separateBuildDir', 'Set to False to disable building in the
separate directory', 'yes'))

g_env = Environment(options = opts,
                    tools = ['default', 'qt', 'CVS', 'swig', 'lex', 'yacc',
omniidl_tool],
                    CVS = [],
                    BINDIR = [],
                    LIBDIR = [],
                    LIBPATH = [],
                    LIBS = [],
                    MODULENAME = '',
                    CXXFILESUFFIX = '.cpp',
                    LEX = 'flex',
                    YACC = 'bison',
                    YACCFLAGS = '-d ')

#####
Sconstruct main script which is responsible to handle the complete build
# like parsing the folder to find sconstruct or to select compiler and flags!
#####
configFile = ARGUMENTS.get('config', 'default')
buildConfig = GlobalConfig(configFile)

# must ensure that we are still in the good folder
source_root = buildConfig.source_root.replace('\\', '/')
build_root = buildConfig.config.get("build", "build_root")
install_root = buildConfig.config.get("build", "install_root")
# this flag is optional
try:
    msvs_ver = buildConfig.config.get("compiler", "MSVS_VERSION")
except:
    msvs_ver = 0
# where to look for headers
try:
    inc_path = [buildConfig.config.get("build", "stlport_include")]
except:
    inc_path = []

inc_path = inc_path + [ source_root,          # normal headers
                        build_root,          # generated stuff is here
                        build_root + '/vhdKernel', # and here
                        source_root          + '/vhdKernel' ] +
buildConfig.config.get("compiler", "INCLUDE_PATH").split() # global (components)

# linker, useful for DISTCC usage, but optional
try:
    linker = buildConfig.config.get("compiler", "LINK")
except:
    linker = None

if linker is not None:
    g_env['LINK'] = linker

# TMP directory
try:
    g_env['ENV']['TMP'] = buildConfig.config.get("build", "tmp_path")
except:
    g_env['ENV']['TMP'] = os.environ.get('TMP', ".")
```

```
#####
Setup builders for optional components
#####
try:
    optional_components = buildConfig.config.get("build",
"optional_components").split()
    g_env.Append(CPPDEFINES = [ (flag, '') for flag in optional_components ])
except:
    optional_components = []

#we would like to append the env variables in order to help scons to find tools
g_env['ENV']['PATH'] = g_env['ENV']['PATH']+os.environ['PATH']
g_env['PLATFORM'] = buildConfig.platform
g_env['SWIG'] = buildConfig.config.get("compiler","SWIG")
#we take the list of optional component as SWIG define
swigDefine=""
for define in optional_components:
    swigDefine = swigDefine + " -D" + define
g_env['SWIGFLAGS'] = buildConfig.config.get("compiler","SWIGFLAGS") + swigDefine
g_env['CC'] = buildConfig.config.get("compiler","CC")
g_env['CXX'] = buildConfig.config.get("compiler","CXX")
g_env['CPPFLAGS'] = buildConfig.config.get("compiler","CPPFLAGS_GLOBAL")
g_env['CXXFLAGS'] = buildConfig.config.get("compiler","CXXFLAGS")
g_env['CFLAGS'] = buildConfig.config.get("compiler","CFLAGS")
g_env['CPPPATH'] = inc_path
g_env['INCDIR'] = inc_path
g_env['LIBDIR'] = buildConfig.config.get("compiler","LIB_PATH").split()
g_env['LIBPATH'] = buildConfig.config.get("compiler","LIB_PATH").split()
g_env['LINKFLAGS'] = buildConfig.config.get("compiler","LDFLAGS").split()
g_env['MSVS_VERSION'] = msvs_ver
# settings used by distcc
g_env['ENV']['DISTCC_DIR'] = os.environ.get('DISTCC_DIR', '')
g_env['ENV']['DISTCC_HOSTS'] = os.environ.get('DISTCC_HOSTS', '')
g_env['ENV']['HOME'] = os.environ.get('HOME', '')
#we create the global macro for dll based on the type of project compiled
if g_env['typeProject'] == 'dynamicLib':
    g_env.Append(CPPDEFINES = ['MAKE_DLL'])

buildConfiguration = buildConfig.config.get("build","configuration_name")

#windows user would like to append d to the libsname in debug i.e. libnamed.lib
if buildConfig.platform == 'win32':
    if buildConfiguration.find('debug') >= 0:
        postLibExtension = '_md_dbg'
    else:
        postLibExtension = '_md'
else:
    postLibExtension = ''
#####
Handling library names
#####def
mapLibNames(key_list):
    return filter(lambda(x): x is not None, map(lambda(x):
buildConfig.libmap.get(x, None), key_list))

#####
Export general settings
#####
def doExport(env):
    Export('env')
    typeProject = "${typeProject}"
    env.Export('typeProject') #could be either staticLib, dynamicLib, program
    env.Export('postLibExtension')
    env.Export('source_root')
    env.Export('build_root')
    env.Export('install_root')
    env.Export('buildConfiguration')
    env.Export('optional_components')
```



```
#####
Parse and Build
#####
# search for each of the subdirectories
def buildLibs():
    if BUILD_TARGETS == []:
        target_list = buildConfig.modules_list
    else:
        target_list = BUILD_TARGETS

    for moduleName in target_list:
        #first we need to do the export of variable from Sconstruct to sub
Sconscript
        env=g_env.Copy()
        doExport(env)

        #we can execute it but first export the module name and others settings
        env.Export('moduleName')

        # link libraries for DLLs and executables
        libs = libsMapped = []
        env.Export('libs')
        env.Export('libsMapped')

        installIncludePath = install_root+'/include/'+ moduleName
        env.Export('installIncludePath')
        installPyPath = install_root+'/py'
        env.Export('installPyPath')
        installIDLPath = install_root+'/idl'
        env.Export('installIDLPath')
        installBinPath = install_root+'/bin'
        env.Export('installBinPath')
        installPDBPath = install_root+'/pdb'
        env.Export('installPDBPath')
        installLibPath = install_root + '/lib'
        env.Export('installLibPath')

        buildDir = build_root + '/' + moduleName
        env.Export('buildDir')

        if env['separateBuildDir']:
            SConscript(moduleName + "/SConscript",
                        exports = 'env',
                        build_dir = buildDir,
                        duplicate = 1)

            env['MODULENAME'] = buildDir
        else:
            SConscript(moduleName + "/SConscript",
                        exports = 'env')
            env['MODULENAME'] = moduleName

        #before executing the real job we enforce some flags that must
        #be read-only so we avoid that a SConscript modify some of the
        #flags in order to ensure better consistency
        env['CC'] = g_env['CC']
        env['SWIG'] = g_env['SWIG']
        env['MSVS_VERSION'] = g_env['MSVS_VERSION']
        env['CXXFLAGS'] = g_env['CXXFLAGS']
        env['CCFLAGS'] = g_env['CCFLAGS']

        #import from Sconscript data such as Source files, libs, ...
        # and invoke compiler, swig, moc,... through sconsbuilt-in fcts
        compileModule(env)
#####
Start build
#####
buildLibs()
```


Appendix J

Posters

The following figures represent a series of posters illustrating some results obtained using the framework describes in this thesis. Theses applications were mostly directly connected to EU projects and were developed by multiple developers principally from VRLab.

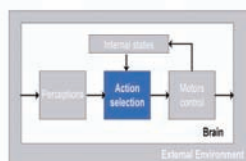


Figure J.1: Cultural heritage.



Action Selection for Autonomous Virtual Humans

Virtual humans can decide what to do on their own in real-time without help of animators according to their motivations and the environment thanks to an action selection model. They could be used to create more interesting NPCs (Non Player Characters) in computer games.

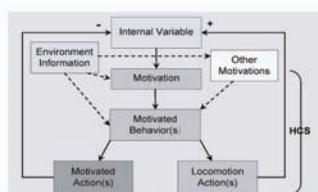


Based on ethology models

An action selection model has to choose at each moment in time, the most appropriate action, out of a repertoire of possible actions. How to apportion one's available time so as to satisfy several needs.



Model and "Virtual Life"



Motivations	Locations	Actions
hunger	table	eat
thirst	sink, table	drink
toilet	toilet	satisfy
resting	sofa	rest
sleeping	bed	sleep
washing	bath	wash
reading	bookshelf	read
playing	computer	play
watering	plant	water
idle (default)	sofa	watch TV



Our model is based on hierarchical classifier systems (HCS) working in parallel (one per motivation), and associated with a free flow hierarchy for the propagation of the activity. A 3D apartment was designed in which the virtual human can "live" autonomously by perceiving his environment and satisfying different conflicting motivations at specific locations.

Results

Our model respects criteria for designing action selection mechanisms based on ethology in particular compromise behaviors. Virtual human always chooses most appropriate action on its own in real time according to its motivations and environment information. Some extensions are planned to complexify the model.

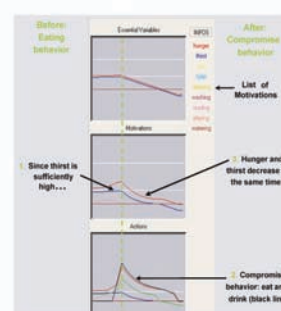


Figure J.2: Illustration of the action selection for autonomous virtual humans from Etienne de Sevin.

Augmented Reality

STAR (EU project)

(Service Training through Augmented Reality)

This project aims to train technicians who need to perform service operations in industrial installations. A virtual environment consisting of virtual humans and objects is displayed together with the real, through use of augmented reality techniques. The virtual humans are mainly intended to guide the user on how a particular service operation can be accomplished.

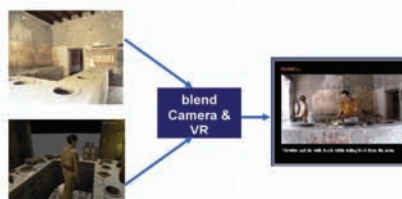


At each step, the application provides the user with a set of choices, consisting of the actions that the virtual human is capable of undertaking. Once the user makes his choice, the virtual human is displayed performing the corresponding action.

Lifepius (EU project)

(Innovative Revival in Ancient Frescos - Paintings and Creation of Immersive Narrative Spaces Featuring Real Scenes with Behavioral Fauna and Flora)

LIFEPLUS is an application of augmented reality and real time 3D graphics applied to archaeological sites. Innovative technologies make possible for the visitor in Pompeii to go back in the past and live again events and experiences that really took place. Ancient people are coming back to life to meet and talk about facts and things of their time.



The system uses a standard firewire camera in order to capture images and blends the virtual environment with them. The tracker does not require any markers allowing to setup the system virtually anywhere. For improving the realism, a significant effort have been produced for generating complex human animations.

Figure J.3: Augmented reality simulations.

VRlab
Virtual Reality Laboratory

ERATO

identification Evaluation and Revival of the Acoustical heritage of ancient Theatres and Odea

INCO-MED 3-years project started in February 2003

This virtual restitution integrates the visual and acoustical simulations of ancient Roman theatres and odea, based on the most recent results of research in archaeology, theatre history, clothing, theatre performance and early music.

VRlab is involved in the creation of the 3D visual model of Aphrodisias Odeon and the animation of the audience. The virtual audience reacts in realtime to the actor's play on stage.

odeon reconstruction

Hi-fidelity actors

virtual human model

color variety texture

color variety example

roman social classes

animation examples

spraying emotions

VRlab team: Zerrin Celebi Ghavami, Rachel Cêtre, Branislav Ulicny, Pablo de Heras Ciechowski, Sébastien Schertenleib, Jonathan Maim, Mireille Clavien - **Research director:** Prof. Daniel Thalmann

Figure J.4: ERATO project overview.

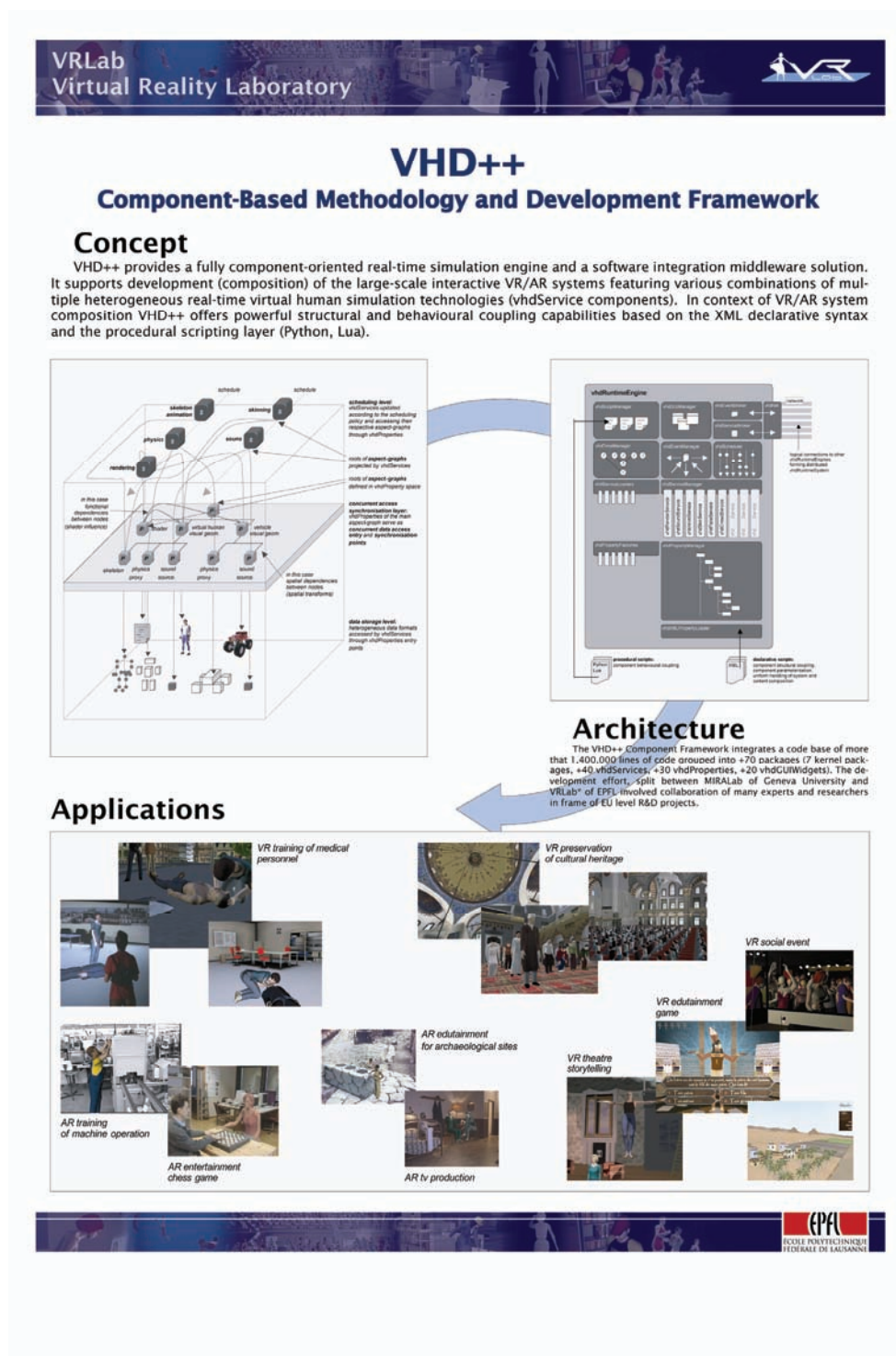
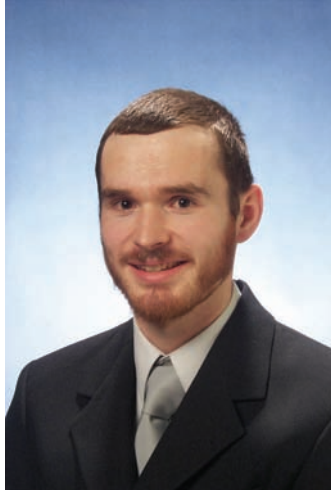


Figure J.5: VHD++ platform overview.

VITA

Sébastien Schertenleib



Sébastien Schertenleib obtained his master degree in computer science in 2002 from the Swiss Federal Institute of Technology in Lausanne (EPFL). He speaks French, English, and German. Currently, he is a research assistant at VRLab (EPFL). He is working on systems architectures for the European Commission funded projects JUST, LIFEPLUS, ERATO, NEWTON, and EPOCH dealing with emergency training, cultural heritage, and crowd simulations. His research interests include multitasking and data-driven architectures for 3DRTS, as well as multi-agent simulations. His interests include studying history, reading fantasy literature, and playing board games and videogames.

Publications:

- Sébastien Schertenleib, Daniel Thalmann, “Managing High-Level Script Execution Within Multithreaded Environments”, in Game Programming Gems 6, Charles River Media, March 2006.
- Sébastien Schertenleib, Daniel Thalmann, “Designing a Multilayer, Pluggable AI Engine”, in Game Programming Gems 6, Charles River Media, March 2006.
- Pablo de Heras Ciechomski, Sébastien Schertenleib, Daniel Thalmann, “Animation the Aphrodisias Odeon – a system overview”, ERATO Symposium, Turkey, 2006
- Barbara Yersin, Jonathan Maïm, Pablo de Heras Ciechomski, Sébastien Schertenleib, Daniel Thalmann, “Steering a Virtual Crowd Based on a Segmentically Augmented Navigation Graph”, VCROWDS, 2005.
- Pablo de Heras Ciechomski, Sébastien Schertenleib, Jonathan Maïm, Daniel Thalmann, “Reviving the Roman Odeon of Aphrodisias: Dynamic Animation and Variety Control of Crowds in Virtual Heritage”, VAST, 2005.
- Daniel Thalmann, Pablo de Heras Ciechomski, Jonathan Maïm, Julien Pettre, Sébastien Schertenleib, Barbara Yersin, “Course 15: Crowd and Group Animation: Real-Time Crowds”, Siggraph, August 2005.
- Pablo de Heras Ciechomski, Sébastien Schertenleib, Jonathan Maïm, Jonathan Maïm, Damien Maupu, Daniel Thalmann, “Real-Time Shader Rendering for Crowd in Virtual Heritage”, 2005.
- Sébastien Schertenleib, Daniel Thalmann, “An Effective Cache-Oblivious Implementation of the ABT Tree“, in Game Programming Gems 5, Charles River Media, March 2005.

- George Papagiannakis, Sébastien Schertenleib, Brian O’Kennedy, Marlene Arevalo-Poizat, Nadia Magnemat-Thalmann, Andrew Stoddart, Daniel Thalmann, “Mixing Virtual and real scene in the site if ancient Pompeii”, *Computer Animation and Virtual Worlds*, p 11-24, Volume 16, Issue 1, February 2005.
- Sébastien Schertenleib, Mario Gutierrez, Frédéric Vexo, Daniel Thalmann, “Conducting a Virtual Orchestra: Multimedia Interaction in a Music Performance Simulator”, *IEEE Multimedia*, Special Issue on Multisensory Communication and Experience through Multimedia, Vol 11, Issue 3, pages 40-49, July-September 2004.
- George Papagiannakis, Sébastien Schertenleib, Michal Ponder, Marlene Arevalo-Poizat, Nadia Magnemat-Thalmann, Daniel Thalmann, “Real-Time Virtual Humans in AR Site”, *IEEE Visual Media Production (CVMP)*, pp 273-276, London, UK, 15-16 March 2004.
- George Papagiannakis, Sébastien Schertenleib, Brian O’Kennedy, Michal Ponder, Nadia Magnemat-Thalmann, Daniel Thalmann, Andrew Stoddart, “Visualizing and Tracking Virtual Humans in AR Cultural Heritage Sites”, *Proceedings of the Workshop on augmented Virtual Reality (AVIR)*, 2003.
- Michal Ponder, Bruno Herbelin, Tom Molet, Sébastien Schertenleib, Branislav Ulicny, George Papagiannakis, Nadia Magnemat-Thalmann, Daniel Thalmann, “Immersive VR Decision Training: Telling Interactive Stories Featuring Advanced Virtual Human Simulation Technologies”, 9th Eurographic Workshop on Virtual Environments IPT/EGV2003, pp 97-106, Zurich, Switzerland, 2003.
- Michal Ponder, Bruno Herbelin, Tom Molet, Sébastien Schertenleib, Branislav Ulicny, George Papagiannakis, Nadia Magnemat-Thalmann, Daniel Thalmann, “Interactive Scenario Immersion: Health Emergency Decision Training in JUST Project”, *VRMHR2002 Conference Proceedings, VRLab-EPFL*, pp 87-101, Switzerland, November 2002.
- Sébastien Schertenleib, “3D Real-Time Sound Rendering: A VHD++ Component using DirectX 8.1”, *Diploma Master Project, EPFL*, March 2002.